Experimental dimer-monomer ^{3}He NMR shift inside C_{60} explained from first principles by Path Integral Monte Carlo modelling

Master's thesis

Markku Alamäki
NMR Research Unit
University of Oulu
2019

Contents

1 Introduction

Fullerenes are molecules formed of carbon atoms. Buckminsterfullerene C_{60} is a fullerene that consists of 60 carbon atoms arranged into a ball formation (figure 1). In the Buckminsterfullerene, each carbon atom has three "neighbours" that are covalently bonded to that atom. The Buckminsterfullerene can be flattened to a graph (figure 2). It is seen that it has $60 \cdot (3/2) = 90$ bonds, 12 pentagonal faces, and 20 hexagonal faces. All the atoms in Buckminsterfullerene are also "equal", so that with appropriate symmetry rotations any two atoms can be transposed. The bonds can also be classified as "hh-bonds" and "ph"-bonds depending on if they bound two hexagonal faces or one pentagonal face and one hexagonal face, respectively.

One or two helium atoms may be contained within Buckminsterfullerene. These kinds of fullerenes that have atoms trapped within them are called *endohedral fullerenes*. If a Buckminsterfullerene has a single helium (monomer) atom trapped inside it is denoted as $He@C_{60}$. If two helium atoms are trapped (dimer) it is denoted as $He_2@C_{60}$.

In NMR spectroscopy, chemical shift describes how much an external magnetic field is attenuated or boosted by the chemical environment of an atom nucleus. Information about the chemical environment can therefore be gained from the NMR spectrum, or alternatively concentrations of chemical compounds can be measured in a sample, which is the idea behind magnetic resonance imaging (MRI).[2]

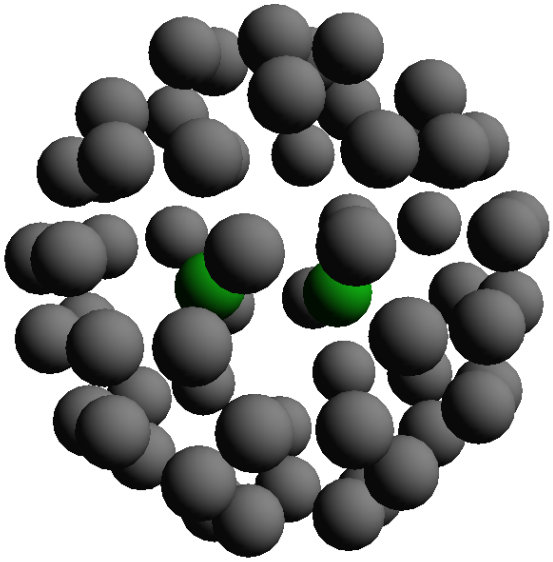

Figure 1: *Buckminsterfullerene C_{60} with two helium atoms inside.*

Systems with cavities in them, such as fullerenes, are often studied through the chemical shifts that they impose upon noble gases, such as helium and xenon[3]. Various theoretical and experimental studies have been performed on endohedral fullerenes, where inert gas molecules are trapped within the fullerene cage[3, 4, 5, 6].

In this thesis, the chemical shift of a helium atom inside the Buckminsterfullerene is studied by means of computer simulations. The fullerene is assumed to contain either one or two helium atoms, which gives two endohedral fullerenes $He@C_{60}$ and $He_2@C_{60}$. Additionally the corresponding hexa-anions $He@C_{60}^{6-}$ and $He_2@C_{60}^{6-}$ are studied.

The systems were simulated in three different numerical model types; a) spherical, b) rigid cage and c) fully dynamic. In the spherical model, only the dynamics of the helium atoms are simulated, while the effect of carbon atoms is represented only as spherically symmetric fields acting on the helium atoms. Only one model (named here as SPHERICAL) used in this thesis is of this type, which is the same model that has been used before[5], and which describes the neutral fullerenes. In the rigid cage type simulations, the carbon atoms are static, and the interaction between helium and carbon atoms is described as pairwise additive fields. Only one model (RIGID) is used of this sort, which simulates the neutral fullerenes. In the fully dynamic models, the carbon atoms are allowed to move according to a potential energy function[1] which considers the angles and distances be-

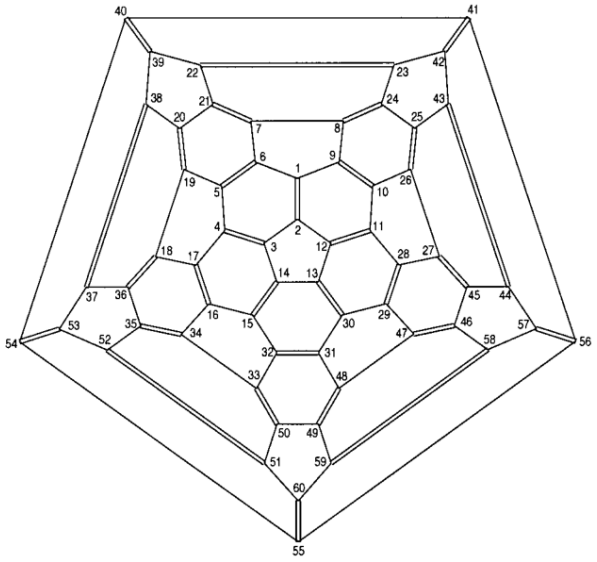

Figure 2: *Buckminsterfullerene "flattened" to a graph. Each node in the graph corresponds to a carbon atom and each edge in the graph corresponds to a bond between the atoms. The single lines correspond to "ph"-bonds, while the double lines correspond to "hh"-bonds. Picture from [1].*

tween various carbon atoms. Three different models of this sort are used, in which only the pairwise additive fields between helium and carbon atoms are varied. The first model (FULL) uses the same fields as the RIGID model, and therefore describes the neutral fullerenes. The second model of this sort (FULL_NEW_NEUTRAL) also describes the dynamics of the neutral fullerenes, but the interaction between helium and carbon atoms is described differently. The last fully dynamic model (FULL_NEW_ANION) models the anion fullerenes.

The simulations are implemented as *path integral Monte Carlo (PIMC)* simulations, which describe the particles spatial location as paths in the three dimensional space, instead of the more usual description as single points in space. These paths are the path integrals from Feynman's formulation of quantum mechanics[7], and their inclusion add a quantum mechanical effect that is associated with the quantum mechanical motion of the particles. This effect changes the spatial particle distributions. The result from a PIMC simulation is similar to a normal classical MC simulation, but with additional small correction to the positions. The Trotter expansion is used to discretize the path integrals, which results in a discretization parameter called *Trotter number*[8, 9], which describes the accuracy of the path sampling, with higher numbers corresponding to finer detail. This Trotter number will be varied

in the simulations to see how quantum mechanics affect the chemical shifts. Separate Trotter numbers will be specified for helium and carbon atoms by using the linear interpolation method described by Li & Miller [10]. This will allow to separately see the effect of using quantum statistics for helium and carbon atoms.

In addition to varying the simulation type, Trotter number and the number of helium atoms, the temperature of the simulations will be varied between 50 K and 300 K, which allows studying the temperature dependence of the chemical shifts. The simulations at 300 K will be compared to experimental results[4].

2 Theory

This chapter outlines the theory behind Monte Carlo (MC) and Path Integral Monte Carlo (PIMC) simulation methods, and how the results from these simulations are interpreted to get estimates on statistical properties of the simulated system.

2.1 Chemical shift

An atomic nucleus with a nuclear spin I will have a nuclear magnetic moment

$$\mu = \frac{\gamma I h}{2\pi}, \tag{1}$$

where γ is a proportionality constant called *magnetocyric ratio* and h is the Planck constant[2]. When the atomic nucleus is placed in an external magnetic field, the energy levels of the nucleus become separated by the energy interval

$$\Delta E = \frac{\gamma h B}{2\pi}, \tag{2}$$

where B is the applied magnetic field[2]. The nucleus can change the energy level by absorbing or emitting a photon with the same energy as this gap. The resonance condition in NMR experiments is

$$\nu = \frac{\gamma B}{2\pi}, \tag{3}$$

where ν is the photon frequency[2]. Various instrumentations can detect this resonance, and by varying either the frequency or the external magnetic field, spectral data can be obtained[2].

The magnetic field experienced by the atomic nucleus may be different from the applied magnetic field. This difference between the two fields is called *nuclear shielding*, and it is proportional to the applied field[2]. The relative difference, i.e. nuclear shielding divided by the applied magnetic field, is called the *chemical shift*[2].

Chemical shifts can be estimated with simulation methods. A straightforward way would be to simulate the molecular system by running a Monte Carlo simulation with the system energy determined by an ab-initio calculation. Then for each state obtained by the simulation, an additional ab-initio calculation could be done to determine the chemical shift experienced by the nucleus of interest. Then the chemical shift would be averaged over all the obtained values to get an estimate of the true value. This method requires of course a lot of computational resources, as a very large number of ab-initio calculations are done. A less computationally extensive way is to calculate effective energy and NMR force fields[1] for the system, that are based on some kind of functions which are fitted to data obtained by ab-initio methods. This is the method that is used in this thesis.

2.2 Density matrix

This subchapter attempts to outline the most important properties of the *density matrix*, or *density operator* representation of a quantum system. The theory of density matrices are discussed for example in [11].

Quantum systems are represented as vectors in a Hilbert space[12]. Assume that we have a quantum system with a Hilbert space $\mathcal{H}$, and that the set $\{|\phi_i\rangle, i = 1, 2, \dots N\}$ is an orthonormal basis of $\mathcal{H}$. A general state in this space is then a linear combination of the basis vectors

$$|\Psi\rangle = \sum_{i=1}^{N} c_i |\phi_i\rangle, \sum_{i}^{N} c_i^2 = 1. \tag{4}$$

When a measurable A of this quantum system is detected, the system randomly goes to one of the eigenstates of the corresponding operator $\hat{A}$. This is the *wave function collapse*. The probability of detecting the eigenstate $|a_k\rangle$ is

$$P(|a_k\rangle) = |\langle \Psi_i | a_k \rangle|^2 \tag{5}$$

Now assume that we have an ensemble of quantum states, so that a random system that is picked from this ensemble can be in the states $|\Psi_i\rangle, i =$

[1] A "NMR force field" just gives a contribution to some chemical shift based on the particle locations. It is therefore not actually a "force" field despite the name.

$1, 2, \ldots M$, that each occur with corresponding probabilities p_i. When the property A of this system is measured, the system will again end up in one of the eigenstates of the operator $\hat{A}$. The probability detecting the system in state $|a_k\rangle$ is now

$$P(|a_k\rangle) = \sum_{i=1}^{M} p_i |\langle \Psi_i | a_k \rangle|^2, \tag{6}$$

which combines the usual notion of probability and wavefunction collapses[12]. The right hand side of this equation can be rewritten;

$$\begin{aligned}
\sum_{i=1}^{M} p_i |\langle a_k | \Psi_i \rangle|^2 &= \sum_{i=1}^{M} p_i \langle a_k | \Psi_i \rangle \langle \Psi_i | a_k \rangle \\
&= \langle a_k | \left(\sum_{i=1}^{M} p_i |\Psi_i\rangle \langle \Psi_i| \right) |a_k\rangle \\
&= \langle a_k | \hat{\rho} | a_k \rangle
\end{aligned}$$

where we have defined the density operator $\hat{\rho}$ in the last equality. Note that this density operator is independent on the choice of measurable. Also all the statistical information about the system is included in this density operator, so it fully describes the system. However, since many different ensembles can result in the same density operator, the full ensemble representation includes redundant information that is not needed to describe the behaviour of the system. On the other hand, the density operator only includes the essential information[11].

2.2.1 Density operator of canonical ensemble

Consider a particle that experiences a potential function V for which the position basis $|x\rangle$ is the eigenbasis. If the particle is connected to a heat bath of some temperature T, the ensemble that describes the statistics is the canonical ensemble. For a system in thermal equilibrium the density matrix is of form [11]

$$\hat{\rho} = e^{-\beta \hat{H}} = e^{-\beta(\hat{T}+\hat{V})},$$

where $\hat{T}$ is kinetic energy operator, and $\beta = 1/(k_{\mathrm{B}}T)$ is the inverse temperature. Next a path integral representation for the quantum mechanical counterpart is derived. For this derivation the density operator of a free particle is needed. In position basis it has the form [8]

$$\rho^{free}(\mathbf{r}, \mathbf{r}'; \beta) = \exp\left(\frac{-m}{2\beta\hbar^2} (\mathbf{r} - \mathbf{r}')^2 \right). \tag{7}$$

Also the Trotter approximation[8, 9]

$$e^{-\beta(\hat{T}+\hat{V})} = \left[e^{-\frac{\beta}{M}\hat{V}} e^{-\frac{\beta}{M}\hat{T}}\right]^M, \tag{8}$$

is needed. Using this approximation and inserting identity operators $\int |\mathbf{r}\rangle \langle \mathbf{r}| \, \mathrm{d}\mathbf{r}$ in between one gets for the density matrix

$$\begin{aligned}
&\rho(\mathbf{r_0}, \mathbf{r_M}; \beta) \\
&= \langle \mathbf{r_0}| \, e^{-\beta(\hat{T}+\hat{V})} \, |\mathbf{r_M}\rangle \\
&\approx \langle \mathbf{r_0}| \left[e^{-\frac{\beta}{M}\hat{V}} e^{-\frac{\beta}{M}\hat{T}}\right]^M |\mathbf{r_M}\rangle \\
&= \int \mathrm{d}\mathbf{r_1} \ldots \mathrm{d}\mathbf{r_{M-1}} \, \langle \mathbf{r_0}| \, e^{-\frac{\beta}{M}\hat{V}} e^{-\frac{\beta}{M}\hat{T}} \, |\mathbf{r_1}\rangle \, \langle \mathbf{r_1}| \, e^{-\frac{\beta}{M}\hat{V}} e^{-\frac{\beta}{M}\hat{T}} \, |\mathbf{r_2}\rangle \\
&\quad \ldots \langle \mathbf{r_{M-1}}| \, e^{-\frac{\beta}{M}\hat{V}} e^{-\frac{\beta}{M}\hat{T}} \, |\mathbf{r_M}\rangle \\
&= \int \mathrm{d}\mathbf{r_1} \ldots \mathrm{d}\mathbf{r_{M-1}} e^{-\frac{\beta}{M}\sum_{n=0}^{M-1} V(\mathbf{r_n})} \, \langle \mathbf{r_0}| \, e^{-\frac{\beta}{M}\hat{T}} \, |\mathbf{r_1}\rangle \, \langle \mathbf{r_1}| \, e^{-\frac{\beta}{M}\hat{T}} \, |\mathbf{r_2}\rangle \\
&\quad \ldots \langle \mathbf{r_{M-1}}| \, e^{-\frac{\beta}{M}\hat{T}} \, |\mathbf{r_M}\rangle \\
&= \int \mathrm{d}\mathbf{r_1} \ldots \mathrm{d}\mathbf{r_{M-1}} e^{-\frac{\beta}{M}\sum_{n=0}^{M-1} V(\mathbf{r_n})} e^{-\frac{mM}{2\beta\hbar^2}(\mathbf{r_0}-\mathbf{r_1})^2} e^{-\frac{mM}{2\beta\hbar^2}(\mathbf{r_1}-\mathbf{r_2})^2} \\
&\quad \ldots e^{-\frac{mM}{2\beta\hbar^2}(\mathbf{r_{M-1}}-\mathbf{r_M})^2} \\
&= \int \mathrm{d}\mathbf{r_1} \ldots \mathrm{d}\mathbf{r_{M-1}} \exp\left[-\beta \sum_{n=0}^{M-1} \left(\frac{V(\mathbf{r_n})}{M} + \frac{mM}{2\beta^2\hbar^2}(\mathbf{r_n} - \mathbf{r_{n+1}})^2\right)\right],
\end{aligned} \tag{9}$$

where $V(\mathbf{r_n})$ is the potential energy in position basis. The first step in the above equation is the definition, the second step follows from the Trotter approximation (8), the third step adds identity operators, the fourth step "moves" the potential energy operators outside the brakets, the fifth step inserts the free particle density operator (7), and the last step reorders the equation. The obtained from in the above equation is a discrete representation of a *path integral*. This is called so because the positions $\mathbf{r_0} \ldots \mathbf{r_M}$ are thought to describe a path of how the particle propagates in the *imaginary time* β (figure 3). The density matrix is often called a *propagator*[8, 13] in this context. Often interesting is the diagonal part of the density operator, which we get by setting $\mathbf{r_0} \to \mathbf{r}$ and $\mathbf{r_M} \to \mathbf{r}$ in equation (9);

$$\begin{aligned}
\rho(\mathbf{r}, \mathbf{r}; \beta) = &\left[\int \mathrm{d}\mathbf{r_1} \ldots \mathrm{d}\mathbf{r_{M-1}} \right. \\
&\left. \exp\left[-\beta \sum_{n=0}^{M-1} \left(\frac{V(\mathbf{r_n})}{M} + \frac{mM}{2\beta^2\hbar^2}(\mathbf{r_n} - \mathbf{r_{n+1}})^2\right)\right]\right]_{\mathbf{r_0}\to\mathbf{r}, \mathbf{r_M}\to\mathbf{r}}
\end{aligned} \tag{10}$$

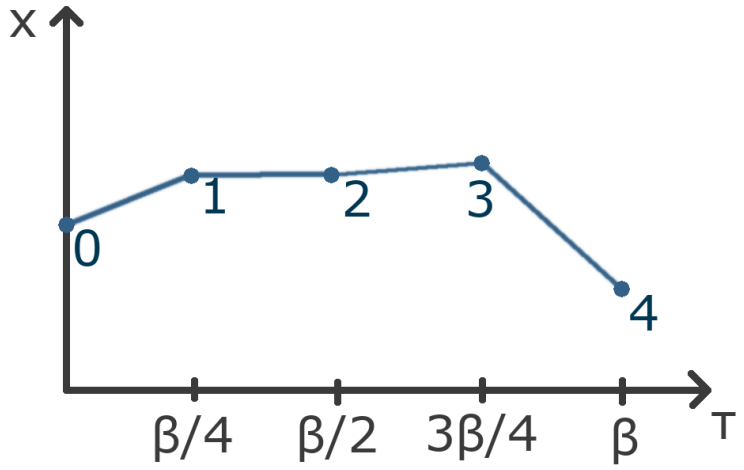

Figure 3: *Discrete representation of path of a particle that moves in 1 dimension. The Trotter number is here 4, which leads to the path being represented with 5 discrete points. The imaginary time τ runs from 0 to β with step size $\beta/4$.*

Because $\mathbf{r_0} = \mathbf{r_M}$, the equation (10) describes paths that end up in the same position as where they started. If the operator of interest has the position basis as its eigenbasis, this diagonal part of density operator is enough to get the statistical spectrum of that observable.

Classical corresponding operator would be

$$\rho^{classical}(\mathbf{r}, \mathbf{r}; \beta) = e^{-\beta V(\mathbf{r})}, \tag{11}$$

which does not depend on the speed of the particle (unless the potential energy depends on speed). Comparing equations (10) and (11) it is seen that the path integral form of the density operator corresponds to a classical density operator of a modified system with M particles if the potential energy experienced by that system is

$$V(\mathbf{r_0}, \mathbf{r_1}, \ldots, \mathbf{r_{M-1}}) = \sum_{n=0}^{M-1} \left(\frac{V(\mathbf{r_n})}{M} + \frac{mM}{2\beta^2\hbar^2} (\mathbf{r_n} - \mathbf{r_{(n+1) \mod M}})^2 \right), \tag{12}$$

and the additional degrees of freedom $\mathbf{r_1}, \ldots \mathbf{r_{M-1}}$ are traced over. The resulting equation is exactly equation (10). It is seen that the equation (12) defines a system of M "beads" which are connected to each other by harmonic potentials to form a "necklace". From now on, the index that enumerates these beads will be called either *"bead index"*, *"slice index"* or *"time slice index"*.

Each of the M beads experiences the potential of the classical system divided by M. Therefore the classical system and the quantum mechanical system correspond to each other exactly when the harmonic potential that binds the beads together is very stiff, i.e. $m/\beta^2 \to \infty$. When the potential is not stiff, the particle is delocalized slightly, which may affect the position distribution of the atom. Typically this delocalization allows the atom to "push" further into any constricting potential walls since the increase in the energy is smaller when only one bead pushes into the potential wall.

If the system contains more than one atom, the equation (12) needs to be adapted. The isomorphic classical system for many particles is one where the all the particles are split into chains of beads like before, and where the classical potential is evaluated over all the resulting potentials separately[13]. Specifically, in the notation that has been used in this chapter so far, the potential is of form

$$V(\mathbf{r_0^0}, \ldots, \mathbf{r_{M-1}^0}; \ldots; \mathbf{r_0^{K-1}}, \ldots, \mathbf{r_{M-1}^{K-1}}) = \sum_{n=0}^{M-1} \left(\frac{V(\mathbf{r_n^0}, \ldots, \mathbf{r_n^{K-1}})}{M} + \sum_{k=0}^{K-1} \frac{m_k M}{2\beta^2\hbar^2} (\mathbf{r_n^k} - \mathbf{r_{(n+1) \mod M}^k})^2 \right), \tag{13}$$

where the new index enumerates the K different atoms. This equation can be justified by thinking the system as a single particle with $3K$ degrees of freedom, and then developing the density operator to a path integral like before in the equation (12). Since the first term $(V(\mathbf{r_n^0}, \ldots, \mathbf{r_n^{K-1}})/M)$ on the right of the equation describes how the system would behave classically, that term can be called the "classical" term. The other term which describes the deviation from classical system can be called a "quantum mechanical" or "kinetic energy" term.

The equation (13) defines multiple copies of the system that are connected to each other. Each of these copies are called a *time slice* of the system. Like before with the beads, these time slices form a kind of a necklace. Each of the single atom beads now feel the potential caused by the other atoms of the same time slice and additionally the harmonic attraction of the corresponding atoms in the neighbouring time slices.

2.3 Monte Carlo simulation

The Monte Carlo (MC) methods[14] use random variables to solve different problems. One class of MC methods is generating random samples from a probability distribution[14]. These random samples in turn can be used to approximate an integral weighted with that probability distribution. This way of integrating is very effective when we have multidimensional integrals,

as the error estimate does not depend on the number of dimensions. Specifically, the precision of the calculated integrals are at worst $O(N^{-\frac{1}{2}}$, where N is the number of datapoints used[14]. In many other methods, if computational time is kept constant, the error term increases as the number of dimensions increases.

2.3.1 Markov chain Monte Carlo simulation

Markov chains[15, 16, 17] are a way of generating random draws from a probability distribution even when the probability distribution is not known, or too difficult to calculate[14]. This is the case when we want to generate draws from a canonical ensemble of many particles, when the calculation of the normalization factor $1/Z$ is too difficult. It is therefore not possible to directly draw states from this distribution. Instead it is possible to construct a process that, when repeated long enough, will simulate the distribution. This is the idea behind Markov Chain Monte Carlo (MCMC)[14, 18, 17] simulation.

The basic idea behind the MCMC methods can be summarized as follows. Although the method is often used with continuous distributions, for simplicity assume that we have a discrete distribution, so that we have a system for which the set of allowed states is the finite set S. Each of the states $a \in S$ has a probability $P(a)$ assigned to them, defining a discrete probability distribution. Denoting the probability of the system moving from state $a \in S$ to state $b \in S$ with $T_{a\to b}$, we construct a stochastic process which satisfies three conditions

1. The process is a *Markov process*, which means that the probability of the system moving to state $a \in S$ only depends on the current state of the system. A Markov chain is a chain of states generated by a Markov process.

2. For all states $a, b \in S$ it holds that $\frac{T_{a\to b}}{T_{b\to a}} = \frac{P(b)}{P(a)}$. This condition is often called *detailed balance*[8, 17].

3. The process is ergodic, meaning that if the process is repeated long enough, the system will traverse all states.

If these conditions hold, the process will simulate the probability distribution; the process defines a chain of states $a_0, a_1, \ldots, a_i \in S$ in which the frequency of the states correspond to the probability distribution that is simulated. Of course, depending on the transition $T_{a\to b}$, consecutive states in the chain may be correlated, but if the process is simulated long enough, the correct probability distribution can be obtained.

2.3.2 Trial moves

An important part of creating a Markov chain Monte Carlo simulation is the implementation of the transitions $T_{a\to b}$. In addition to satisfying the three conditions for correctly simulating the probability distribution, the implementation of the transitions should be so that

1. It requires low computational resources.
2. It produces fast convergence of the distribution, i.e. the states in the chain are not too correlated.

Both of these requirements deal with the total computational time required to get a converged position distribution. After all, the total time required is simply the time required per step times the number of steps required.

In the best case scenario we can generate direct draws from the probability distribution, in which case we can define the transition probability as

$$T_{a\to b} = P(b),$$

so the consecutive states will be entirely uncorrelated. This kind of simulation is of course usually not very useful, since the position distribution is already known and often the whole point of the simulation is to find the position distribution. When the position distribution is not known, the transition probability is often represented as a product of two numbers;

$$T_{a\to b} = C_{a\to b}A_{a\to b}, \tag{14}$$

where $C_{a\to b}$ is *suggestion probability* and $A_{a\to b}$ is *acceptance probability*. The transition is implemented so that the probability distribution $C_{a\to b}$ is used to choose a candidate state, so that when the system is in state a, the probability of choosing b as the candidate is $C_{a\to b}$. This candidate state will then be accepted with probability $A_{a\to b}$. If the candidate state is accepted, it becomes the next state in the chain, otherwise the previous state is retained. This procedure causes the effective transition probability to be as defined in the equation (14).

With this suggestion and accept/reject method the detailed balance condition becomes

$$\frac{C_{a\to b}A_{a\to b}}{C_{b\to a}A_{b\to a}} = \frac{P(b)}{P(a)} \tag{15}$$

This method is usually how the transitions are implemented. This equation still leaves a lot of freedom in choosing the suggestion and acceptance probabilities. This choice should be done with computational time in mind, and what is effective depends on what kind of system is simulated.

2.3.3 Path Integral Monte Carlo

In a *Path Integral Monte Carlo* (PIMC) simulation the Markov chain Monte Carlo method is used to simulate path integrals. If the goal is to get the diagonal part (diagonal in position basis) of the system, one defines the equivalent classical system as in equation (13). One then simulates this equivalent classical system and records the state of the system in the zeroth time slice[2]. The other time slices do not matter since it is the zeroth slice that defines the position distribution. The only reason the other time slices are used in the simulation is that they affect the position distribution of the zeroth slice.

Once we have defined the system in the form (13), we need to decide what kind of trial moves will be used. Choices include the simple *single slice moves*, *bisection moves* and different rotations and displacements. Several moves can be also combined into a single move, since if all the moves satisfy the detailed balance condition, the combination move will also satisfy the detailed balance condition. One could for example have the combination move include trial moves to all the atoms separately. Alternatively a number of different, randomly selected trial moves can be used, as long as the transition probability of the combined move stays constant. This kind of implemention allows a lot of freedom in selecting how often the different trial moves will be executed, which is useful if we want to balance CPU-time evenly between the trial moves.

2.3.3.1 Single slice move

In a single slice move, a single bead is moved[9]. This means that the suggested next state is the exact copy of the previous state, except one of the beads will be displaced by a small amount. Algorithmically this choice works as follows;

1. Select a random time slice and atom, this defines a single bead.
2. Generate a displacement of that bead from a probability distribution function $\mathrm{pdf}(x, y, z)$. The probability distribution function should be symmetric with respect to each argument. This symmetry requirement implies $C_{a\to b} = C_{b\to a}$.

The acceptance probability will be $A_{a\to b} = \min\left(\frac{P(b)}{P(a)}, 1\right)$, which causes the detailed balance condition (15) to be true.

[2]Alternatively any time slice could be recorded, since the system is symmetric with respect to rotating the time slices.

The width of the probability distribution function used for generating displacements is usually adapted so that e.g. around 50% of trial moves are rejected. This is because too wide displacement will cause the moves almost never to be accepted, which leads to slow convergence. If too narrow displacements are used, the trial moves are almost always accepted but the atoms move very slowly, which too leads to slow convergence.

Since to execute a single slice move one only needs to calculate the change in potential energy generated by the external potential function within the time slice and the energy from the bead-bead interaction between the two neighbouring beads, execution of this kind of a trial move can be made fast. However, the single slice move becomes increasingly poor choice when the number of time slices is increased. With increasing number of slices, the coupling between the adjacent beads becomes stronger, and therefore smaller movements of the beads lead to large increases of energy. This means that if the displacement size is optimized for 50% acceptance, the displacements need to become smaller and smaller, which leads to slow diffusion.

2.3.3.2 Bisection move

Bisection moves[9, 8] are specifically crafted for PIMC. They work by moving a chain of $2^k - 1$ beads at the same time. The number k is called the *level* of the move[9]. The move is constructed so that the old state of these beads is completely forgotten. If we apply this move to a chain of beads where the bead index runs from 0 to 2^k, it works in k steps:

1. The position of the 2^{k-1}:th bead is selected from a 3D-Gaussian distribution $f(\mathbf{r}, \mu, \sigma) = (2\pi\sigma^2)^{-\frac{3}{2}} \exp\left(\frac{-(\mathbf{r}-\mathbf{r_{middle}})^2}{2\sigma^2}\right)$, where $\mathbf{r_{middle}}$ is the arithmetic mean of locations of 0:th and 2^k:th beads and the variance $\sigma^2 = \frac{\hbar^2\tau}{2m}$. Here $\tau = \frac{2^{k-1}}{M}\beta$ is the amount of imaginary time between 2^k time slices.

2. If $k = 1$ the algorithm has finished. If $k > 1$ this algorithm is recursively repeated two times, first for beads with indices 0 to 2^{k-1} and then with beads with indices 2^{k-1} to 2^k.

This recursive algorithm perfectly samples the density matrix of a free particle, so that the acceptance probability will be 100% for free particles. If we have a particle with interactions, the acceptance will be

$$A_{a\to b} = \min(1, e^{-\beta(V_b - V_a)}), \tag{16}$$

where V_a and V_b are the energies resulting from the classical part of potential. The chain length 2^k can be adapted to improve the acceptance ratio of this

method. However, it should be noted that, although lowering the chain length lowers the computational cost of a single step and improves acceptance, it also causes the consecutive states to be more alike. Therefore the optimal solution may not be to optimize to 50% acceptance, but instead an even lower acceptance (e.g. 5%) may sometimes work better.

This method can further be improved with *multilevel Metropolis method*[9, 19], which evaluates the acceptance simultaneously while the new chain is being constructed, allowing unlikely paths to be rejected earlier. This lowers the computational cost of rejected steps considerably, and therefore it becomes more desirable to use higher level moves at the expense of lower acceptance ratio.

2.3.3.3 Modified bisection move

If the atoms are a part of a molecule with very stiff bonds, the normal bisection move may not work well. This is especially true in low temperatures where the inverse temperature β is large.[3] The bisection method suggests moves that are too large compared to the scale of the classical part of potential. On the other hand using normal single slice moves would also be very slow.

A simple solution that works at low temperatures and is better than single slice moves is to use moves that moves multiple beads at the same time, but by a smaller amount than the real bisection move. The solution works by constructing a Brownian bridge from $(0,0,0)$ to $(0,0,0)$, using the same kind of recursive algorithm as with the normal bisection move. This Brownian path is then applied to the selected chain of beads as a displacement (as opposed to replacing the positions) to construct the suggested move. Specifically, if the move is applied to a chain of length 2^k+1, with the bead index i running from 0 to k, the displacement of each bead, denoted as $\Delta\mathbf{r_i}$ is constructed as follows. First the displacement of the first bead and last bead is set to zero; $\Delta\mathbf{r_0} = \Delta\mathbf{r_k} = 0$. Next the following recursive algorithm is used to determine the rest of the displacemen

1. The displacement of the 2^{k-1}:th bead is selected from a 3D-Gaussian distribution $f(\mathbf{r},\mu,\sigma) = (2\pi\sigma^2)^{-\frac{3}{2}} \exp\left(\frac{-(\mathbf{r}-\mathbf{r_{middle}})^2}{2\sigma^2}\right)$, where $\mathbf{r_{middle}}$ is

[3] At low temperatures the kinetic energy term in (13) becomes small compared to the classical term, causing the coupling between time slices become looser compared to the inter-atomic coupling. In this kind of system the moves suggested by the bisection method are almost always rejected. In cold temperatures it would be maybe better to also rotate and translate the rigid molecules as a whole. However, often one wants to increase the Trotter number for lower temperatures, making the normal bisection method again more favorable.

the arithmetic mean of displacements (as opposed to locations) of 0:th and 2^k:th beads and the variance $\sigma^2 = c\tau$. Here $\tau = \frac{2^{k-1}}{M}\beta$ is the amount of imaginary time between 2^k time slices, and the proportionality factor c can be freely selected, for example to optimize the acceptance ratio.

2. If $k = 1$ the algorithm has finished. If $k > 1$ this algorithm is recursively repeated two times, first for beads with indices 0 to 2^{k-1} and then with beads with indices 2^{k-1} to 2^k.

Then the displacement is added to the current locations of the beads, which defines the suggested trial move. Since this defines probability distribution function for the move that is symmetric (as with single slice moves), the acceptance probability will be $A_{a\to b} = \min(P(b)/P(a), 1)$, which is the same as for single slice moves.

In the same way as with single slice moves, the displacement size can be optimized, so this method can be thought of as a generalization of single slice moves. Since there are two parameters that can be optimized (chain length and move size), they can not both be optimized by requiring acceptance ratio of 50%. In test simulations longest possible chain length appeared to be the best choice.

This method is actually the result of my misunderstanding of how the real bisection move works. Although I later implemented the correct bisection move too, this modified version proved to be useful in some cases. This method may not be the optimal way to displace a chain of beads since it was an accident, but it is relatively simple and since it is very similar to the normal bisection move, it can easily be implemented if the normal bisection move is implemented, with minimal changes to source code.

2.4 Time series analysis

As explained before, a PIMC simulation produces a chain of states, that is the positions of the atoms in the first time slice. If one calculates a value of an observable x for each of the collected states, one gets a discrete series of that value $x_0, x_1, \ldots, x_{n-1}$. If the simulation was run long enough before collecting the datapoints and the simulation style (e.g. the trial moves) remain unchanged, this series can be considered as a part of a *stockhastic stationary process*[4]. Such processes can be analyzed with the methods described in the

[4]In a stationary process the system can be thought to be in some kind of a stationary state e.g. a thermal equilibrium. A stationary process can be contrasted with *evolutionary processes*, in which the statistical properties of the process change somehow.

following subchapters. The notation in the following is partly based on the book by Priestley [20] which is a comprehensive book on time series analysis.

2.4.1 Mean, variance, skew, kurtosis

The mean of the process is

$$\mu = E[x_t]. \tag{17}$$

It can be estimated by *sample mean*;

$$\overline{x} = \frac{1}{n}\sum_{i=0}^{n-1} X_i. \tag{18}$$

The variance is

$$\sigma^2 = E[(x_t - \mu)^2] \tag{19}$$

It can be estimated by *sample variance*

$$s^2 = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - \overline{x})^2. \tag{20}$$

Standardized moment of k:th, $k = 1, 2, \ldots$ degree is

$$\mu_k = \frac{1}{n\sigma^k}\sum_{i=0}^{n-1}(x_i - \mu)^k. \tag{21}$$

These moments can be estimated in the same as mean and variance. Skew is the third standardized moment and kurtosis is the fourth.

2.4.2 Fourier transforms

Forward finite Fourier transform X_n of x_k is

$$X_n = \sum_{k=0}^{N-1} x_k e^{-i2\pi kn/N}. \tag{22}$$

Inverse finite Fourier transform x_k of X_n is

$$x_k = \frac{1}{N}\sum_{n=0}^{N-1} X_n e^{i2\pi kn/N}. \tag{23}$$

2.4.3 Autocorrelation

Autocovariance of the process is

$$R_\tau = E[\{X_t - \mu\}\{X_{t+\tau} - \mu\}],$$

which can be estimated with the *biased estimate*;

$$\hat{R}_\tau = \frac{1}{N} \sum_{t=0}^{N-\tau-1} (x_t - \overline{x})(x_{t+\tau} - \overline{x}) \tag{24}$$

Autocorrelation is autocovariance scaled by variance;

$$\rho_\tau = \frac{R_\tau}{\sigma^2},$$

which can be estimated with the other estimates as

$$\hat{\rho}_\tau = \frac{\hat{R}_\tau}{s^2}. \tag{25}$$

2.4.4 Integrated autocorrelation time

Integrated autocorrelation time (IAcT) measures the total amount of correlation between the datapoints. It is defined[5] [22] as

$$\tau_{\text{int}} = \frac{1}{2} + \sum_{\tau=1}^{\infty} \rho_\tau.$$

It can be estimated by using the estimate $\hat{\rho}_\tau$ and truncating the summation at some point

$$\tau_{\text{int}} = \frac{1}{2} + \sum_{t=1}^{n_{\text{cutoff}}} \hat{\rho}_\tau, \tag{26}$$

where the truncation point n_{cutoff} should be much smaller than the total data size N, since the accuracy of the estimate $\hat{\rho}_\tau$ drops rapidly as $\tau \to N$. The cutoff point can not of course be too small either since otherwise the long "tail" part of the autocorrelation function is ignored. A more sophisticated method is to use a "lag window"[21] instead of a cutoff. In the lag window method the IAcT is estimated as

$$\tau_{\text{int}} = \frac{1}{2} + \sum_{t=1}^{N-1} w_t \hat{\rho}_\tau,$$

where N is the number of datapoints and w_t is a weighing factor that goes to zero as $t \to N$. Therefore all available data is used in the estimation, but the less accurate data is given less weight.

[5] Some authors instead define IAcT as twice as large.[21].

2.4.5 Statistical inefficiency

Statistical inefficiency[23] also measures the total amount of correlation in the data. Consider a stationary stockhastic process $A_0, A_1, \ldots$ that has a mean $\mu(A)$ and variance $\sigma(A)$. The block averaging of the process is the process $B_0^K, B_1^K, \ldots$

$$B_k^K = \frac{1}{K} \sum_{k=0}^{K-1} A_{kN+1}, \tag{27}$$

where K is the *reduction factor*. The block averaging therefore divides the series into K "blocks", each containing the same number of datapoints, and then calculates the average value inside each block. If the original process is Gaussian (meaning that the corresponding distribution is Gaussian), the process B is also Gaussian with the same mean. The variance of the resulting distribution will be

$$\sigma^2(B^K) = \frac{s(K)}{K} \sigma^2(A), \tag{28}$$

where $s(K)$ is a function that will depend on how correlated the data is. Statistical inefficiency is the limit

$$s = \lim_{K \to \infty} s(K) = \lim_{K \to \infty} \frac{K \sigma^2(B^K)}{\sigma^2(A)}. \tag{29}$$

If $s = 1$ then the original series is assumed to have no correlation, while a higher value means that there is correlation.

Statistical inefficiency can be estimated from a discrete series $A_0, A_1, \ldots, A_{N-1}$ by using the block averages that can be calculated from that limited data; $B_0^K, B_1^K, \ldots, B_{\lfloor N/K \rfloor}^K$. The variances in equation (29) are approximated with sample variances, and the limit is approximated by replacing K with some large enough number K^{large}. The resulting approximation is

$$s \approx \frac{K^{\text{large}} \frac{1}{\lfloor N/K^{\text{large}} \rfloor} \sum_{i=0}^{\lfloor N/K^{\text{large}} \rfloor} (\mu - B_i^{K^{\text{large}}})^2}{\frac{1}{N} \sum_{i=0}^{N-1} (\mu - A_i)^2}, \tag{30}$$

where μ is the sample average calculated from the series A. Here $K^{\text{large}} << N$ is required, since otherwise there are too few terms in the sum that approximates $\sigma^2(B^K)$. If this condition can not be fulfilled, a better approximation may be obtained by calculating the approximation with multiple different K:s. The estimate should be based on calculations done with intermediate block sizes, since using the smallest block sizes effectively ignore the long "tail" part of autocorrelation function. On the other hand the estimates calculated with biggest block sizes are inaccurate. This compromise between

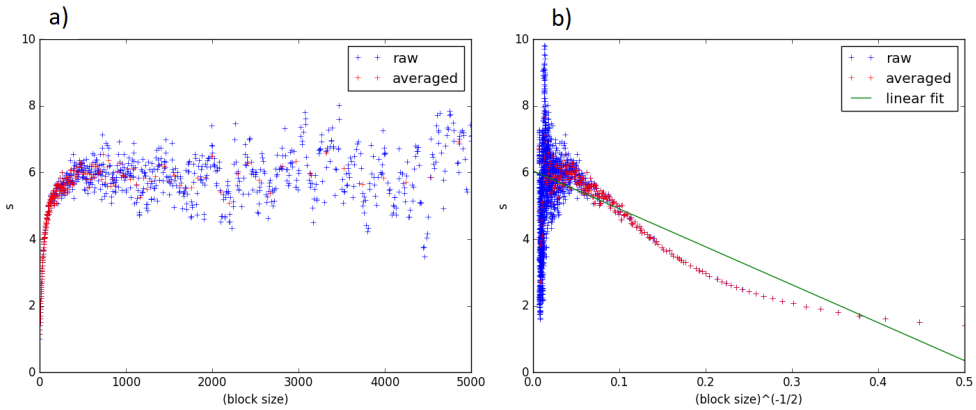

Figure 4: *Approximation of statistical ineffieciency s with the equation (30). Length of the time series is 100000. The "raw" datapoints are directly calculated from equation (30), and the "averaged" data are averaged from the "raw" data. (a): It is seen that the value of s reaches a "plateau" at approximately when block size reaches* 500*. The limit at infinity can be approximated to be the height of this plateau, which seems to be approximately six. (b): The x-axis has been changed to the inverse square root as suggested in [23]. A line "linear fit" has been fitted to the "averaged" data. By checking the intersection point of this line with y-axis statistical inefficiency can be automatically approximated.*

small/large block sizes is the same compromise that is done with cutoff in the estimation of IAcT.

One good way to get a weighted average of multiple estimates is to plot the approximations against block sizes and then fit a line to this data with proper weighting. In picture 4 is an example of this procedure.

2.4.6 Standard error of the mean

When the mean is estimated with sample mean, there is always some error. The standard error of the mean (SEM) measures the size of this error. If there is no autocorrelation, SEM is calculated as

$$\text{SEM} = \sqrt{\frac{\sigma^2}{N}}, \tag{31}$$

where N is the number of datapoints and σ^2 is the variance[23]. If the data is correlated, this estimate needs to instead use "effective size" of the data. Because of the correlation, the successive datapoints are very similar and

"contain" partly the same information. The SEM is now calculated from

$$\text{SEM} = \sqrt{\frac{\sigma^2}{N_{\text{effective}}}}, \tag{32}$$

where $N_{\text{effective}}$ is the effective data size. The effective data size can be calculated from the IAcT τ_{int}, or from statistical inefficiency s. If calculated from the statistical inefficiency, the number of datapoints needs to be divided by the statistical inefficiency[23]. If the IAcT method is used, the error estimate is scaled with the squareroot of two times the IAcT[22], i.e. the effective data size is divided by IAcT times two. This can be summarized as

$$N_{\text{effective}} = \frac{N}{2\tau_{\text{int}}} = \frac{N}{s}. \tag{33}$$

From this equation it is seen that if the IAcT and statistical inefficiency are correctly calculated, the sanity condition

$$2\tau_{\text{int}} \approx s, \tag{34}$$

should be fulfilled. This can be used to evaluate reliability of the result. If τ_{int} and s give vastly different result for the effective size, there probably is not enough data (or the data is too correlated) to reliably calculate τ_{int} or s. On the contrary if τ_{int} or s give approximately the same result, it is a good indication that the SEM will be correctly estimated, although it can never quarantee it.

The reliability of estimation of both τ_{int} and s is challenging for basically the same reason. Namely, if the autocorrelation function has a long "tail", meaning that if we write the autocorrelation function as a spectrum[6] of exponential functions

$$\rho_\tau = \int_0^\infty c(x) e^{-\tau x} \, \mathrm{d}x, \tag{35}$$

the spectrum contains significant components at small values of the parameter x. This equation defines the ρ_τ as the Laplace transform of the spectrum $c(x)$. Noting that a Laplace transform resembles a Fourier transform with a imaginary frequency, the x in the above equation will in the following text be called "*imaginary frequency*". With this terminology, low imaginary frequencies correspond to a long tail in the autocorrelation function.

[6] In the context of thermal transient testing, similar spectra are called "time constant spectra"[24].

Consider approximating the equation 26 with an integral;

$$\tau_{\text{int}} = \int_{t=0}^{n_{\text{cutoff}}} \hat{\rho}(\tau)\,\mathrm{d}\tau, \tag{36}$$

where $\hat{\rho}(\tau)$ is calculated from $\hat{\rho}_\tau$ by some kind of interpolation, and additionally the tail of the autocorrelation function is assumed to be so long that the constant term $1/2$ can be ignored. Inserting the spectrum representation (35) to (36), we get

$$\begin{aligned}
\tau_{\text{int}} &= \int_{t=0}^{n_{\text{cutoff}}} \int_0^\infty c(x)e^{-\tau x}\,\mathrm{d}x\,\mathrm{d}\tau \\
&= \int_0^\infty \int_{t=0}^{n_{\text{cutoff}}} c(x)e^{-\tau x}\,\mathrm{d}\tau\,\mathrm{d}x \\
&= \int_0^\infty \frac{c(x)}{x}\left[1 - e^{-n_{\text{cutoff}}x}\right]\,\mathrm{d}x
\end{aligned}$$

The second term within the brackets represents the error generated by truncation since it vanishes when we let $n_{\text{cutoff}} \to \infty$. To get an intuiton of what is the effect of cutoff on the error, consider a simple spectrum consisting of a single imaginary frequency, in which case the spectrum $c(x) = \delta(x - x_0)$, where $x_0 > 0$. Now the above approximation becomes

$$\begin{aligned}
\tau_{\text{int}} &= \int_0^\infty \frac{\delta(x - x_0)}{x}\left[1 - e^{-n_{\text{cutoff}}x}\right]\,\mathrm{d}x \\
&= \frac{1}{x_0}\left[1 - e^{-n_{\text{cutoff}}x_0}\right]
\end{aligned}$$

As it was assumed before that the autocorrelation function has a long tail, the above approximation only holds for small x_0. We see that in order for this approximation to be accurate, we need to have $e^{-n_{\text{cutoff}}x_0} \approx 0$, which in turn means that a lower value of x_0 (that is, a long tail in autocorrelation) requres a higher cutoff.

In order to be able to estimate IAcT correctly, enough datapoints need to be collected. This amount depends on how small imaginary frequencies there are in the corresponding spectrum. If we only have some number of datapoints collected, the lowest imaginary frequencies are always undetectable by numerical analysis. Some kind of background knowledge of the simulated process is therefore required in order to decide how many datapoints is enough.

3 Fullerene molecule models

In this thesis five different computational models will be used; SPHERICAL, RIGID, FULL, FULL_NEW_NEUTRAL and FULL_NEW_ANION models. The following subchapters describe these models.

All the models include a set of force fields. The chemical shifts are approximated by a shift field/NMR force field for chemical shifts, which gives the chemical shift as a function of the atom positions. The constants of the force fields are defined so that length is in Ångstroms and energy is in Kelvins scaled by the Boltzmann constant. The constants of the NMR force fields also use Ångstroms. **These unit conventions apply to all numerical constants given in this chapter.**

3.1 SPHERICAL model

The SPHERICAL-model will in this thesis mean the model that was described and simulated in [5]. This model approximates the fullerene molecule as spherically symmetric cavity, in which the helium atoms move. To construct the force field, the energy of the endohedral fullerene $He@C_{60}$ was calculated in various configurations using ab initio methods.[7] The configurations were such that the carbon atoms remained in fixed positions, while the helium atom was moved around. Therefore energy was only a function of the position of the helium atom, which was used to get a spherically symmetric force field of form

$$V_{\text{SPHERICAL}}(r) = A_2 r^2 + A_4 r^4 + A_6 r^6, \tag{37}$$

with the constants given in table 1.

The interaction between two helium atoms in the dimer ($He_2@C_{60}$) case was described as a force field that is a function of distance between the two helium atoms. This force field too, was produced by using ab initio methods. The energy was calculated at different configurations[8], meaning that the

[7] Datasets produced by Michal Straka at MP2 level using aug-cc-pVDZ basis set.

[8] Datasets produced by Michal Straka at MP2 level using aug-cc-pVDZ basis set.

Table 1: *Force field parameters in spherical cavity model of C60 from[5], to be used with eq. (37).*

A_2	397.763
A_4	259.816
A_6	366.918

Table 2: *Parameters of helium-helium force field from [5], to be used with eq. (38).*

B_6	$-1.83915 \cdot 10^5$
B_7	$2.75932 \cdot 10^6$
B_8	$-1.62169 \cdot 10^7$
B_9	$4.37807 \cdot 10^7$
B_{10}	$-5.30363 \cdot 10^7$
B_{11}	$2.37466 \cdot 10^7$

Table 3: *NMR force field parameters to the spherical NMR force field[5], to be used with eq. (39).*

A	-3.47804
p_0	-0.0015786
p_1	-0.0497113
p_2	0.30585
p_3	-0.527615
p_4	0.292282

distance between the two helium atoms was varied, and then a function was fitted to the obtained data, giving the following force field:

$$V^{\text{He-He}}(r) = \sum_{i=6}^{11} B_i \frac{1}{r^i}, \tag{38}$$

where r is the distance between the two helium atoms, and the constants are given in table 2. These *Extended Lennard Jones (ELJ)* forms[25] are used to define many force and NMR force fields.

The chemical shifts were approximated in much the same way. The chemical shift of helium atom in $He_1@C_{60}$ was calculated[9] at different configurations, which produced the NMR force field

$$\delta^{\text{He-C-SPHERICAL}} = \frac{A}{(r^{\text{He}})^{(\sum_{i=0}^{4} p_i (r^{\text{He}})^i)}}, \tag{39}$$

where r^{He} is the position of the helium atom and the constants are given in table 3. The shift produced by interaction between the helium atom in the dimer case was calculated[10] at different configurations, which gave the

[9] Datasets produced by Michal Straka at B3LYP level of DFT using iglo-3 basis set.

[10] Datasets produced by Perttu Lantto at CCSD(T) level using the aug-cc-pV5Z basis set.

Table 4: *NMR force field parameters for He chemical shift in He dimer.[5], to be used with eq. (40).*

c_6	4.35223
c_8	−50.1457
c_{10}	223.978
c_{12}	−246.725

ELJ-fit

$$\delta^{\text{He-He}} = \frac{c_6}{r^6} + \frac{c_8}{r^8} + \frac{c_{10}}{r^{10}} + \frac{c_{12}}{r^{12}}, \tag{40}$$

where r is the distance between the helium atoms, and the coefficients are given in table 4.

3.2 RIGID model

This model is based on the same ab initio calculations as the SPHERICAL model. The RIGID model uses the same potential energy and NMR force fields for the interaction between helium atoms as SPHERICAL model, but helium-carbon interaction is described differently. Instead of using spherically symmetric fields, the following form is used to describe the interaction between a helium atom and fullerene

$$V_{\text{Int}} = \sum_{i=1}^{60} f^{\text{ELJ}}\left(\sqrt{(\mathbf{r}^{\text{He}} - \mathbf{r}_i^{\text{C}})^2}\right), \tag{41}$$

where $\mathbf{r}^{\text{He}}$ is the position of the helium atom and $\mathbf{r}_i^{\text{C}}$ is the position of i:th carbon atom. The carbon atoms are assumed to rigidly stay in their equilibrium positions defined by the condition that the bond lengths correspond to the experimental result [26] $r_{hh} = 1.401$ Å and $r_{ph} = 1.458$ Å. The function f^{ELJ} is an Extended Lennard-Jones form, whose constants are optimized to match the experimental data. Pertto Lantto made several fits for the ELJ forms, of which the fits given in tables 5 and 6 are used in the RIGID model.

Table 5: *NMR force field parameters for ELJ form of He-C interaction in C_{60}, to be used with equation (41) and an ELJ form similar to eq.(38), where the exponents match with the lower indices of the constants.*

c_6	$-3.57337694935994E+03$
c_7	$2.79642961390445E+04$
c_8	$-7.01435935218870E+04$
c_9	$3.48770389598182E+04$
c_{10}	$5.87207946215140E+04$
c_{11}	$1.39662970481701E+04$
c_{12}	$-3.40006447303070E+04$
c_{13}	$-5.78654710107184E+04$
c_{14}	$-5.95767407262726E+04$
c_{15}	$-4.95542293524539E+04$
c_{16}	$-3.63495760285408E+04$
c_{17}	$-2.45083576996905E+04$
c_{18}	$-1.55282012678113E+04$
c_{19}	$-9.40295363804536E+03$

Table 6: *Energy force field parameters for helium-carbon interaction, to be used with equation (41) and an ELJ form similar to eq.(38), where the exponents match with the lower indices of the constants.*

C_6	$+2.35920520565829E+06$
C_7	$-2.08556180087883E+07$
C_8	$+5.49593304400239E+07$
C_9	$-2.73798446972154E+07$
C_{10}	$-4.42110243257504E+07$
C_{11}	$-8.32374358093650E+06$
C_{12}	$+2.84798459140703E+07$
C_{13}	$+4.57597105441796E+07$
C_{14}	$+4.59302623536588E+07$

3.3 FULL model

The FULL model is similar to the RIGID model, except the carbon atoms are allowed to move around. The energetics of carbon atoms are described by the extended Wu force field[1]. This force field is partly adapted so that the bond lengths match the RIGID model. In this force field, potential energy V is given by

$$\begin{aligned} 2V = {} & c_1 \sum_{30} (\Delta r_h)^2 + c_2 \sum_{60} (\Delta r_p)^2 \\ & + c_3 r_p r_h \sum_{120} (\Delta \alpha_h)^2 + c_4 r_p r'_p \sum_{60} (\Delta \alpha_p)^2 \\ & + 2c_5 \sum_{120} \Delta r_p \Delta r_h + 2c_6 \sum_{60} \Delta r_p \Delta r'_p \\ & + 2c_7 r_p \sqrt{r_h r'_p} \sum_{120} \Delta \alpha_p \Delta \alpha_h + 2c_8 r_h \sqrt{r_p r'_p} \sum_{60} \Delta \alpha_h \Delta \alpha'_h \\ & + 2c_9 \sum_{60} (\Delta R_p)^2 + 2c_{10} \sum_{120} (\Delta R_h)^2, \end{aligned} \tag{42}$$

where r_h denote the length of a hh-bond, r_p denote the length of a ph-bond, R_h denote the distance between "second degree" neighbouring atoms where the "bond" between them is over a hexagon, R_p denote the distance between "second degree" neighbouring atoms where the "bond" between them is over a pentagon, α_p are the pentagon angles and α_h are the hexagon angles, as depicted in the figure 5. Here the Δ denote difference from equilibrium value; e.g. $\Delta r_p = r_p - r_p^e$. The summations run over all the bonds/angles. The equilibrium values are given in table 8. The force field parameters $c_1, \ldots c_{10}$ are given in table 7.

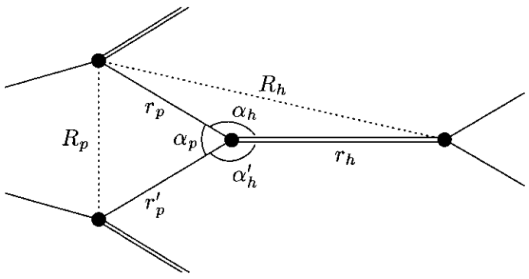

Figure 5: *The parameters used in the extended Wu force field. Picture from [1]*

Table 7: *Constants in the Extended Wu force field* (N/m)*[1].*

c_1	520.1	c_2	348.1
c_3	68.4	c_4	77.6
c_5	28.0	c_6	9.8
c_7	35.1	c_8	21.7
c_9	42.7	c_{10}	11.6

Table 8: *Equilibrium values of the extended Wu force field parameters. The bond lengths r_p^e and r_h^e are the same bond lengths[26] that were used in the RIGID model. The angles α_p^e and α_h^e are exact, and the "bond" lengths R_p^e and R_h^e can be calculated from r_p^e and r_h^e.*

r_p^e	$2.755213\, r_{\mathrm{Bohr}} = 1.458$ Å
r_h^e	$2.647500\, r_{\mathrm{Bohr}} = 1.401$ Å
R_p^e	$\frac{r_p}{2}(1+\sqrt{5})$
R_h^e	$\sqrt{r_p^2 + r_h^2 + r_p r_h}$
α_p^e	$\frac{3}{5}\pi$
α_h^e	$\frac{4}{6}\pi$

3.4 NEW models

This chapter describes two models that will be dubbed "FULL_NEW_NEUTRAL" and "FULL_NEW_ANION". These models are excactly the same as the FULL model, except the ELJ forms for helium-carbon interaction are changed. For these two models, new ab initio calculations and fits were done by Petr Stepanek and Perttu Lantto[27].

The energy was calculated with spin component scaled (SCS) MP2 method with aug-cc-pV5Z[28] basis set on helium atom and def2-TZVPP basis set[29] on carbon atoms, shown previously to provide high quality results[30]. Resolution of identity for Coulomb integrals (RI-J) was used in Hartree-Fock step. The interaction energy $E_{\mathrm{He\text{-}C_{60}}}$ corrected for the basis set superposition error was calculated as

$$E^n_{\mathrm{He\text{-}C_{60}}} = E^n_{\mathrm{He@C_{60}}} - E_{\mathrm{He@C_{60}(ghost)}} - E^n_{\mathrm{He(ghost)@C_{60}}} \tag{43}$$

where fragments labelled (ghost) contain atoms with associated basis set functions, but neither charge nor electrons, and superscript ^{n} denotes the charge of the system (0 or 6-). Note that the system He@C_{60}(ghost) is neutral and identical in both cases as the six additional electrons in the case of anion reside on the fullerene cage.

Table 9: *Energy force field parameters for helium-carbon interaction of FULL_NEW_ANION model, to be used with equation (41) and an ELJ form similar to eq.(38), where the exponents match with the lower indices of the constants.*

C_6	$+1.86093848003356E+05$
C_8	$-7.47270095134864E+06$
C_{10}	$+8.29585875155242E+07$
C_{12}	$-3.39581069702880E+08$
C_{14}	$+5.11791498816027E+08$

Table 10: *Energy force field parameters for helium-carbon interaction of FULL_NEW_NEUTRAL model, to be used with equation (41) and an ELJ form similar to eq.(38), where the exponents match with the lower indices of the constants.*

C_6	$+2.47780438208248E+05$
C_8	$-9.14171146064048E+06$
C_{10}	$+1.00344403867905E+08$
C_{12}	$-4.15991262241875E+08$
C_{14}	$+6.35716691031239E+08$

The chemical shift δ_{He} was calculated with a single helium atom in vacuum used as a reference as

$$\delta^n_{\mathrm{He}} = \sigma_{\mathrm{He@C_{60}}} - \sigma^n_{\mathrm{He@C_{60}}} \tag{44}$$

since the basis set correction in this case was negligible. The individual values of nuclear shielding were calculated in DALTON[31] using BHandHLYP[32, 33, 34] DFT functional and aug-pcSseg-4/def2-TZVPP[35, 29] for helium and carbon atoms, respectively. All calculations were performed in vacuum.

The used parameterizations are given in tables 9, 10, 11 and 12.

Table 11: *NMR force field parameters for helium-carbon interaction of FULL_NEW_ANION model, to be used with equation (41) and an ELJ form similar to eq.(38), where the exponents match with the lower indices of the constants.*

c_6	$-3.86049461794094E+04$
c_7	$2.93275897971387E+05$
c_8	$-8.04409541209000E+05$
c_9	$7.42986394510906E+05$
c_{10}	$3.34207045783956E+05$
c_{11}	$-4.47632121885345E+05$
c_{12}	$-5.46557846855075E+05$
c_{13}	$-1.30158883777584E+05$
c_{14}	$3.54203520514658E+05$
c_{15}	$6.56315628068244E+05$

Table 12: *NMR force field parameters for helium-carbon interaction of FULL_NEW_NEUTRAL model, to be used with equation (41) and an ELJ form similar to eq.(38), where the exponents match with the lower indices of the constants.*

c_6	$-3.22216274521274E+03$
c_7	$2.31435652208172E+04$
c_8	$-5.92157686420225E+04$
c_9	$4.79005951885034E+04$
c_{10}	$3.37791538250778E+04$
c_{11}	$-5.27670989824775E+04$

4 Technical implementation of the calculations.

This chapter describes the programs used to run the PIMC simulations and analyse the results.

The calculations are implemented using three programs; simulation, post-processor and a time series analysis program. The simulation program will be called either *pimc.exe* or *pimc.x*, the post processor program will be called *postproc.exe* or *postproc.x* and the time series analysis program will be called *ts_ana.exe* or *ts_ana.x*. The programs are used as follows;

1. First pimc.x program is used to run a simulation, the raw data (atom positions saved at constant interval) is saved to a file, e.g. *testfile*.
2. The postproc.x is called on the testfile, producing a number of time series files, e.g. *testfile_*<*tsn*>*.ts*, where <tsn> is the name of the time series.
3. The ts_ana.x is called on one of the time series files to produce three files; *testfile_*<*tsn*>*.ana*, *testfile_*<*tsn*>*.ana_ac*, *testfile_*<*tsn*>*.hist*.

This toolchain is depicted in the picture 6. The following three chapters will take a look at each part of this toolchain.

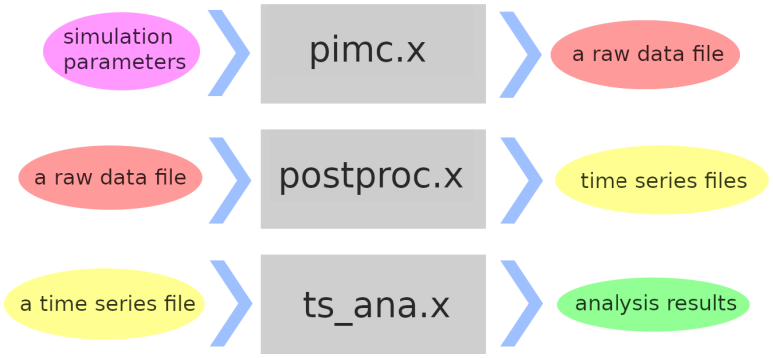

Figure 6: *The tool chain.*

4.1 Simulation program

The simulation program *pimc.x* is the program that runs the actual simulation and produces the raw data. The program uses atomic units in all calculations. If other units are used they are converted to atomic units as soon as possible.

4.1.1 Input

The input is taken from the command line in the form of eight command line arguments that are in order:

1. Temperature in kelvins. Must be a floating point value e.g. "300.0", where the ".0" is required. Using "300" will not work correctly.

2. Trotter number for the helium atoms. Must be of form 2^n, e.g. "8" or "32" are ok while "7" is not.

3. Trotter number for the carbon atoms. Must be of form 2^n.

4. Number of helium atoms. "1" or "2"

5. Number of datapoints. This is how many datapoints will be collected. For example to collect a million datapoints use "1000000".

6. Simulation type. An integer determining what kind of simulation will be run. E.g. "0". See the simulation types below for options.

7. Simulation time. This is approximately how many seconds the simulation will take. For example if simulation time is "3600", the simulation will run for one hour. Larger simulation time allows the program to run more trial moves per datapoint, giving statistically better results.

8. Output filename. The program will save the results in this file. See chapter 4.1.2 below for details.

It should be noted that the program will not check for errors in the input so if the input does not match the above requirements, the program will probably crash, or at least does not produce any meaningful output.

There are a total of six possible simulation types;

- `FULL`, (id. 0) This will simulate the whole system (both helium atoms and carbon atoms).

- `RIGID`, (id. 1) This is the same as "FULL" except the carbon atoms will not move.

- `SPHERICAL`, (id. 2) This simulates the helium atoms in an effective spherically symmetrical potential.

Table 13: *Raw data file header structure.*

bytes	datatype	contains
0-15	character array	Version name.
16-23	double precision float	temperature
24-27	signed integer	helium Trotter number
28-31	signed integer	carbon Trotter number
32-35	signed integer	number of helium atoms
36-39	signed integer	datapoints
40-43	signed integer	simulation type
44-47	signed integer	simulation time
48-	double precision float	raw simulation data

- `C70_RIGID`, (id. 3) This is the same as "RIGID" except the fullerene structure has been modified to approximately match C70 fullerene; the fullerene geometry was split in two and additional atoms were added in between of the two halves. This structure has not been optimized in any way, and the pair potential between helium and carbon may not have been optimized for use case. This simulation type should probably not be used.

- `FULL_NEW_NEUTRAL`, (id. 4) Same as "FULL", except the pair potential form between helium and carbon atoms is changed.

- `FULL_NEW_ANION`, (id. 5) Same as "FULL", exept the pair potential form between helium and carbon atoms is changed.

All of these simulation types have been discussed in more detail in chapter 3, except `C70_RIGID`, which will not be used in this thesis.

4.1.2 Output

If the input to the program was given correctly, the program proceeds to initialize and run the requested simulation, writing the results to a output file at the same time. The output file consists of a 48 byte header followed by the raw simulation data. The header structure is given in the table 13. The raw simulation data consists of the requested number of datapoints saved one after another. Each datapoint consist of all the atom positions saved one after another, and each atom position is represented as three double precision floating point values that give the x, y and z co-ordinates of the atom. Here the atom position means the position of the first bead, so only the first time slice is saved. The positions of helium atoms are always saved before carbon

atom positions. If the dynamics of carbon atoms are not simulated (depends on the selected simulation type), the carbon positions are not saved.

4.1.3 Inner structure

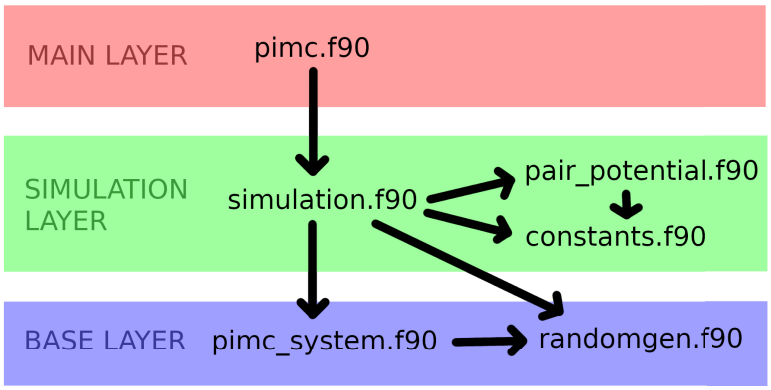

Figure 7: *Simulation program structure. The arrows represent dependencies between the modules.*

The structure of the pimc.x program is depicted in the picture 7. The program consists of six files, that can be divided into three layers; *main-*, *simulation-* and *base-layer*.

4.1.3.1 Base layer

The base layer consists of two files; pimc_system.f90 and randomgen.f90. The randomgen module contains different random number generators that all call directly or indirectly the `random_number` procedure defined in the Fortran 95 standard[36]. The pimc_system module defines a framework for creating PIMC simulations. This module contains no assumption about the simulated system except that it consists of a finite number of atoms that are coupled by some kind of potential energy function. It handles all the work related to trial moves, their optimization and deciding how many times they should be called. Only the information about the molecular system needs to be filled in.

To use the module, the atoms in the simulated system must be divided into groups. It is required that atoms with different mass are not in the same group. Once this kind of division is decided, the potential energy is described as a sum of multiple potentials. For example, if we have a system with two

atom groups, group A and group B, the potential energy could be divided into three parts;

1. Interaction 1 that describes the interactions between the atoms inside the group A.

2. Interaction 2 that describes the interactions between the atoms inside the group B.

3. Interaction 3 that describes the interactions between the two atom groups.

This kind of division of the potential energy into three contributions will allow CPU time savings when the trial moves are evaluated. For example if one atom inside the group A is moved, only the interactions 1 and 3 need to be evaluated. Also when the atoms are divided into groups, the module will allow a different Trotter number for each group, which may also save time if some atoms are "more" quantum mechanic and require a higher Trotter number due to lower mass. This method of using a different Trotter number for different atoms is described in [10]. The basic idea is to place "ghost beads" between the beads to make the Trotter numbers effectively match (see Figure 8). Because the bead indices correspond to different imaginary times, depending on the used Trotter number, it is useful to make a distinction between bead index and time slice index. Hence, the bead index will from now on mean the indexing of a single atom, which runs from 0 to $M-1$, where M is the Trotter number used for the atom. The time slice index (or just slice index), on the other hand, will be the indexing used by the atom that has the highest Trotter number. For example, if we have a two particle system where one particle has a Trotter number 32 and the other particle has a Trotter number 256, the time slice index will run from 0 to 255. There is only one slice indexing for a simulation, but multiple bead indexings.

To create a simulation using the pimc_system module, the subroutine `pimc_system_init` must be called;

```
subroutine pimc_system_init(beta_, ngroups_, groups_,
    nint_, interactions_, time_per_datapoint)
```

The first five arguments describe the system that is to be simulated, and the last argument describes how much time should be used by the simulation. More precisely, the arguments are;

1. `beta_`: The inverse temperature of the system. (double precision)

2. `ngroups_`: Number of atom groups. (integer)

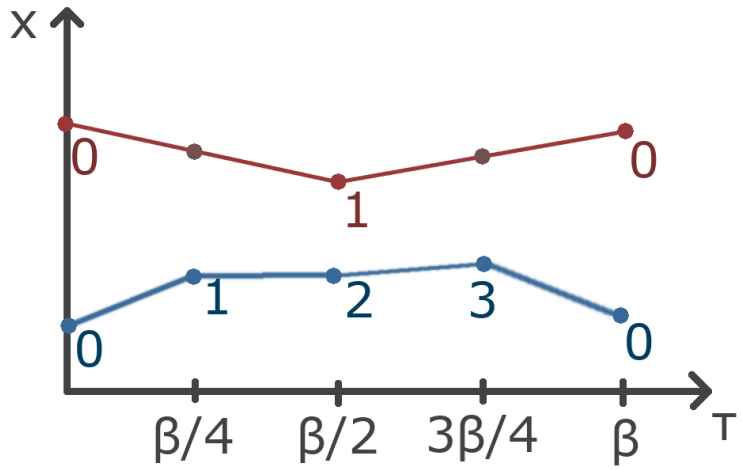

Figure 8: *If the particles of the system are represented using different Trotter numbers, the interactions between them can be evaluated by placing "ghost beads" between them. The red/upper particle uses Trotter number 2 while the blue/lower particle uses Trotter number 4. Here the bead indices 0 and 1 of the red particle matches to the bead indices 0 and 2 of the blue particle. The bead indices 1 and 3 are without a corresponding red pair, however. This problem is circumvented by placing virtual red beads by using linear interpolation.*

3. `groups_`: Array of the `AtomGroup` datatype. The entries of this array correspond to the atom groups described earlier. The place of an atom group in this array will be called *`group_index`*. This index therefore runs from 1 to `ngroups_`.

4. `nint_`: Number of interactions.

5. `interactions_`: Array of the `interaction` datatype. The entries of this array correspond to the interactions in the system. The place of an interaction in this array is called *`interaction_index`*. This index therefore runs from 1 to `nint_`.

6. `time_per_datapoint`: The amount of CPU-time that should be used per datapoint.

Before the initialization routine is called, the `groups_` array and `interactions_` array should be initialized to contain description of the atom groups and interactions, respectively.

Initializing the interactions To initialize the interactions, the "`eval`" procedure pointers

```
...
    type interaction
        procedure (interaction_interface), pointer, nopass
            :: eval
...
```

in them, should be initialized to an function that implements the *interaction_interface* interface:

```
...
    interface
        function interaction_interface(group_index,
            atom_index, slice_index, r_new)
            double precision :: interaction_interface
            integer, intent (in) :: group_index, atom_index
                , slice_index
            double precision, dimension(3), intent (in) ::
                r_new
        end function interaction_interface
    end interface
...
```

The function should calculate the energy difference generated when the bead with the specified `group_index`, `atom_index` and `slice_index` [11] are moved to the place specified by `r_new`. This energy difference should be returned **unscaled** by the Trotter number, and should only include the contribution arising from the interaction that is to be evaluated. To get the positions of the atoms in the time slice that is evaluated, the `get_r` function

```
...
    function get_r(group_index, atom_index, slice_index)
...
```

should be used. This function automatically does the linear interpolation (Fig. 8) required when different Trotter numbers are used.

Initializing the Atom groups The `AtomGroup` data structure

```
...
    type AtomGroup
```

[11] Because in Fortran the array indices start at 1, the `slice_index` also starts at 1 which is different than what has been used earlier in the text. This does not really matter since the handling of the time slices is abstracted away by the pimc_system module.

```
        character(len=40) :: name
        integer :: trotter
        double precision :: mass
        integer :: nat
        double precision, dimension(:,:,:), pointer ::
            r_values
        integer :: nint
        integer, dimension(:), allocatable :: interactions
        double precision :: dx_max
....
```

requires multiple fields to be initialized. The `name` attribute is what the atom group will be called. It is used when printing information about the trial move optimizations. The `trotter` attribute should be initialized to an integer of form 2^n, which will be the Trotter number used for the group. The `mass` attribute is simply the mass of the atoms and `nat` is the number of atoms in the group. The `r_values` pointer must be initialized to point to an array that contains the state of the system. This array is allocated and initialized by the user, and the format of the array is such that the first index enumerates the atoms in the group, the second index enumerates the time slices, and the last index enumerates the three spatial coordintes. The `nint` is the number of interactions the atom group is subject to, and the attribute `interactions` contains the interaction indices of those interactions. Finally, the `dx_max` denotes the size of trial moves. Although it will be automatically optimized, it should be initialized to a reasonable value.

Running the simulation When the `pimc_system_init` is called, it selects a trial move type for each atom group. The possible types are; single slice moves, bisection moves and modified bisection moves. The selected trial move type is then optimized somehow (e.g. by selecting `dx_max`). After each atom group has the trial move defined, the average CPU time consumed by that trial move is estimated. This information allows the trial moves to be called so that each atom group gets the same amount of CPU time, and the total amount of trial moves per datapoint can be selected so that the requested amount of CPU time is consumed per datapoint. Since the CPU time is evenly distributed, the undesirable situation where one trial move type consumes most of the CPU time is avoided. Finally the results of these optimization steps are printed. The simulation can then be run by calling the `relax_system` procedure, that runs the trial moves that were specified by `pimc_system_init`. After each call of `relax_system`, the state of the system should be read by the user from the `r_values` arrays, which

yields one datapoint.

4.1.3.2 Simulation layer

In the simulation layer, the framework provided by the base layer is used to define multiple different simulations. This layer consist of three files; constants.f90, pair_potential.f90 and simulation.f90.

The constants.f90 contains different constants. This includes the extended Wu force field parameterization and the structure of the fullerenes, as described in chapter 3. The structure of the C60 fullerene is described by two data structures; `FULLERENE_R_VALUES` and `C_NEIGHBOURS`. The `FULLERENE_R_VALUES` is an array that describes the equilibrium positions of the 60 carbon atoms. The `C_NEIGHBOURS` describes the bonding between the carbon atoms, listing the neighbours and the neighbours of neighbours of each carbon atom. This information is needed to evaluate energy changes in the extended Wu Force field. The file pair_potential.f90 contains the potential energy functions. The simulation.f90 file defines the actual simulations for each simulation type, two procedures is provided; initialization and running procedures. The initialization procedure initializes a simulation while a running procedure should be called between each datapoint collection.

4.1.3.3 Main layer

The main layer consists of only a single file pimc.f90. This file handles reading the input, initializing and running the correct simulation, and writing the results to disk. It uses the simulation procedures from the simulation layer to implement five different simulation types. The simulation types are FULL, SPHERICAL, C70_RIGID, FULL_NEW_NEUTRAL and FULL_NEW_ANION.

4.2 Postprocessing

In the post processing step, the raw data files from simulations are processed into time series files. Multiple different time series are produced:

1. `he_dist` The distance of the first helium atom from center of mass of the fullerene.

2. `c_dist` The distance of a carbon atom from center of mass of the fullerene. The carbon atom with atom index 1 is tracked. Since the carbon atoms in C_{60} are simililar, it does not matter which atom is tracked.

3. `he_he_shift` The chemical shift imposed upon the first helium atom by the second helium atom (if present). If only one helium atom is present, this time series will not be produced.

4. `he_c_shift` The chemical shift imposed upon the first helium atom by all the carbon atoms in the system.

5. `total_shift` The sum of `he_he_shift` and `he_c_shift`.

Each produced time series file contains the value of the corresponding function evaluated for each datapoint. The time series obtained this way is saved in a simple format; the values are saved sequentially to a binary file as double precision floating point numbers. No header is used.

This program needs to be modified every time we want to produce new kind of time series, so there are multiple versions that each can handle different input file types and produce different time series files. The newest version of this program that is given in appendix does not support some of the time series that have been calculated for this thesis, but instead supports only the current best chemical shift functions.

4.3 Time series analysis program

The time series analysis program analyzes the time series files, producing three files; given a time series file <*filename*>*.ts*, the following files are created:

1. <*filename*>*.ana*, containing a dictionary in JSON format. In this dictionary, six keys are defined:

   - `mean`: The mean of the time series calculated from equation (18).
   - `variance`: The variance of the series calculated from (20).
   - `skew`: The skewness calculated from an approximated form of equation (21).
   - `kurtosis`: The kurtosis calculated from an approximated form of equation (21)
   - `integrated_ac_time`: The estimated integrated autocorrelation time, calculated from equation (26). The cutoff n_{cutoff} is set to a fixed constant value of 1000, so the program will not work for small data sizes.
   - `statistical_inefficiency`: The estimated statistical inefficiency.

2. <*filename*>*.ana_ac*: This is a binary file containing the autocorrelation (equation (25)).

3. <*filename*>*.ana_hist*: This is a JSON formatted file, containing a histogram of the time series.

The statistical inefficiency is approximated using the method suggested earlier (Figure 4). The block averages required for this method are calculated according to the method described in chapter 4.3.1. The autocorrelation is calculated by using Fourier transforms as described in chapter 4.3.2. This program uses the JsonCpp library[37] for writing JSON formatted files, and the FFTW library[38] for calculating Fourier transforms.

4.3.1 Efficient calculation of block averages

Calculation of the block averages for all possible block sizes is a computationally intensive task. A direct calculation, that separately calculates each block averaging takes $O(N^2)$ time where N is the amount of datapoints. This computation can be made more efficient by noting that the block averaging with reduction factor 4 can be alternatively done by first block averaging with a factor 2 and then again with a factor 2. The block averaging with reduction factor 8 can in turn be easily calculated from the data that has been reduced by a factor of 4.

The reduction factors can be represented as a product of prime numbers so that the smallest primes are first in the product $(2^{n_2} \cdot 3^{n_3} \cdot 5^{n_5}, ...)$. For example the number 24 would be represented as $2 \cdot 2 \cdot 2 \cdot 3$. With this representation, each block averaging with factor $p_1, p_2, \ldots, p_n$ (where p_k are prime numbers) can be easily calculated from the block averaged data with factor $p_1, p_2, \ldots, p_{n-1}$. Therefore the calculation of all the block averages whose prime factorisation contains only small primes (e.g. ≤ 5) is relatively cheap to calculate. This calculation can be implemented as a recursive algorithm. Figure 9 shows an example of how recursion could be implemented if one wants to calculate the block averages with reduction factors ≤ 30 whose prime factorization only contains primes ≤ 5. Therefore the reduction factors 7, 11, 13, 14, 17, 19, 21, 23, 26, 28 and 29 are excluded.

If the block averagings are done for the estimation of statistical inefficiency, it does not matter that the reduction factors with large primes in their factorisation are ignored; a good estimation only needs a sufficient number of samples. Since the expensive operations of calculating block averages with large reduction factors are left out, the calculation can be very fast. Although it is not easy to calculate the time complexity of this algorithm, the time saving seems to be substantial, since for example the block averagings

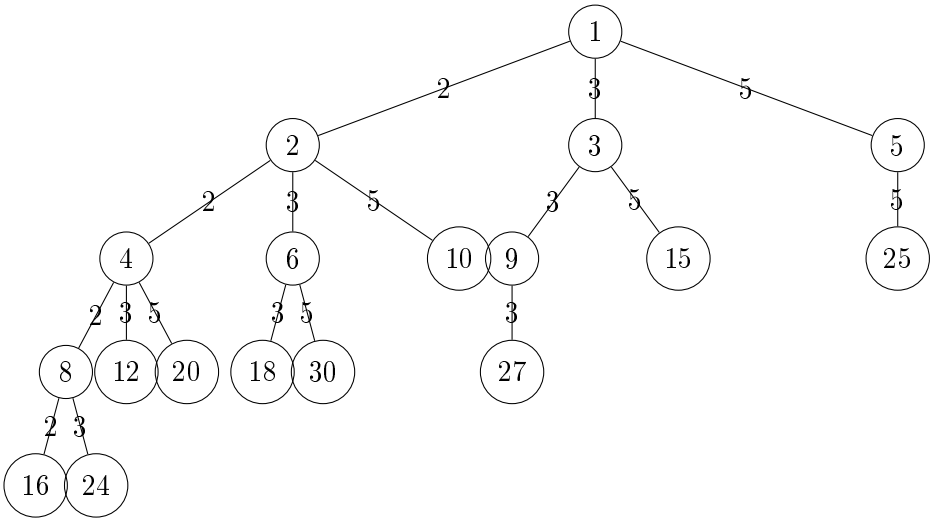

Figure 9: *All integers ≤ 30 whose prime factorization only contains primes ≤ 5 arranged to a tree. Calculating multiple block averages can be implemented as a recursive algorithm whose recursion tree matches one-to-one to the one pictured. The nodes of the tree correspond to the reduced data, and the numbers on the nodes correspond to the reduction factor. Denoting the data with reduction factor x as D_x, the algorithm first starts with the original data D_1, which is reduced to D_2, D_3 and D_5. The D_2 is in turn processed to give D_4, D_6 and D_{10} by reducing with factors 2, 3 and 5, respectively. This is continued until all the required data is obtained.*

required to produce the plots in figure 4 were completed in less than one second, compared to the full calculation, which would take a much longer time. It is also useful to calculate the variances of the block averaged data sets at the same time the block averagings are done, so that the block averaged data sets can be deleted when they are no longer needed to calculate other block averaged data sets. Otherwise the algorithm will use a lot of memory.

4.3.2 Efficient evaluation of autocovariance

If the biased estimate (24) for autocovariance is calculated with direct summation, it will execute in $O(N^2)$ time, which is usually too slow. It is well known that Fourier transforms (22, 23) can be used to calculate this estimate fast[20, 23]. In this chapter I try to provide an explanation of how and why this well known method works.

For convenience let's define the Fourier transforms as functions that take functions as input and return functions. First we denote unnamed functions with arrow notation, for example $x \mapsto |x|$ denotes a function that yields the

absolute value of its argument. The data that will be transformed can be denoted by a map from indices k to the corresponding values x_k; $k \mapsto x_k$. Now we can write the forward Fourier transform for N datapoints $\mathcal{F}_N$ as

$$\mathcal{F}_N[k \mapsto x_k] = \{n \mapsto \sum_{k=0}^{N-1} x_k e^{-i2\pi kn/N}\}$$

The inverse transform will be correspondingly denoted by $\mathcal{F}_N^{-1}$

$$\mathcal{F}_N^{-1}[n \mapsto x_n] = \{k \mapsto \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{i2\pi kn/N}\}$$

Next we define, for a function f, the vectorized form $\mathrm{Vec}_N[f]$, as

$$\{\mathrm{Vec}_N[f]\}[k \to x_k] = \{k \to f(x_k)\}, \tag{45}$$

where N determines the domain of the resulting vectorized function so that it takes integers from the range $[0, N-1]$. Therefore if the input function $(k \to a_k)$ is thought of as a list, the vectorized form of f can be used to apply f to all values in the list separately.

With the above definitions, I define *looped autocovariance* as the function;

$$\mathcal{R}_N^{\text{looped}} = \{\mathcal{F}_N^{-1} \circ \mathrm{Vec}_N(x \to |x^2|) \circ \mathcal{F}_N\}, \tag{46}$$

where $\circ$ denotes function composition. The looped autocovariance function therefore first performs forward Fourier transform, then the resulting data is passed through function $x \to |x^2|$ and finally inverse Fourier transform is performed. Based on the definitions of $\mathcal{F}_N$ and $\mathcal{F}_N^{-1}$ we get

$$\mathcal{R}_N^{\text{looped}}[i \mapsto x_i] = \left\{ k \mapsto \frac{1}{N} \sum_{n=0}^{N-1} \left| \sum_{s=0}^{N-1} x_s e^{-i2\pi sn/N} \right|^2 e^{i2\pi kn/N} \right\} \tag{47}$$

where the right hand side expression can be simplified;

$$\mathcal{R}_N^{\text{looped}}[i \mapsto x_i] = \left\{ k \mapsto \sum_{s=0}^{N-1} \left[x_s^* x_{(s+k) \mod N} \right] \right\}, \tag{48}$$

The summation in this version resembles the summation in the definition of biased estimate of autocovariance. The only differences are;

1. The input data x_i has not been mean corrected.

2. The summation runs longer, with the index "$s+k$" starting again from the beginning.

3. There is a complex conjugation x_s^*. This does not matter if the input series is real valued.

4. There is no scaling with $1/N$

All these problems can be easily rectified. The last problem simply requires us to scale our data at any point by that factor. The solution to the first problem is to simply add an additional step of mean correction to the beginning so that the pipeline becomes;

$$\mathcal{R}_N^{\text{looped}} \circ \mathcal{F}_N^{\text{mean correction}}. \tag{49}$$

The solution to the problem of the summation running too long is to add $N-1$ zeroes to the end of the data, so that the data size becomes $2N-1$. These extra zeroes cause the "extra" terms in the summation to go to zero. The new pipeline is;

$$\mathcal{F}_{2N-1,N}^{\text{resize}} \circ \mathcal{R}_{2N-1}^{\text{looped}} \circ \mathcal{F}_{N,2N-1}^{\text{resize}} \circ \mathcal{F}_N^{\text{mean correction}}, \tag{50}$$

where the new $\mathcal{F}_{k,n}^{\text{resize}}$ operation resizes the data length from k to n by discarding datapoints from the end or adding zeroes to the end. This pipeline calculates the biased estimate exactly (with the exception of the missing $1/N$ factor). If this pipeline is implemented in a programming language, with the data being represented as a list, all the operations will run in linear time, except the Fourier transforms that run in $N \log N$ time. So the total time complexity of this kind of implementation would be $N \log N$, which is better than calculating the autocovariance directly.

5 Results and analysis

Using the simulation program, multiple simulation sets were run. A summary of the simulations is given in the table 14. A total of five simulations sets were run. Within each simulation set, the temperatures, number of helium atoms and Trotter numbers, were varied, creating around one hundred simulations per set. The simulation time was initially set to 10 hours, and then incresed to 100 hours for some simulations. Depending on the simulation parameters, the number of trial moves was somewhere between 10^8 and 10^{12}.

Table 14: *Summary of the simulations. In all simulations, one million datapoints was collected. The number of helium atoms (N_{He}) was either* 1 *or* 2*. Temperatures ranged between* 50 K *and* 300 K*. Different combinations of Trotter numbers for helium (P_{He}) and carbon (P_C) atoms were used.*

simulation set	sim. types.	N_{He}	temp. (K)	(P_{He},P_{C})	sim. time (CPU time)	number of simulations in set
1	SPHERICAL	1,2	50, 100, 150, 200, 250, 300	(1,-), (2,-), ..., (1024,-)	10h	132
2	RIGID	1,2	50, 100, 150, 200, 250, 300	(1,-), (2,-), ..., (1024,-)	10h	132
3	FULL	1,2	50, 100, 150, 200, 250, 300	(1,1), (2,1), ..., (128,1)	10h	96
4	FULL	1,2	50, 100, 150, 200, 250, 300	(1,1), (2,2), ..., (128,128)	10h	96
5	FULL_NEW_NEUTRAL, FULL_NEW_ANION	1,2	50, 100, 150, 200, 250, 300	(1,1), (32,1), (32,32)	100h	72

5.1 SPHERICAL model

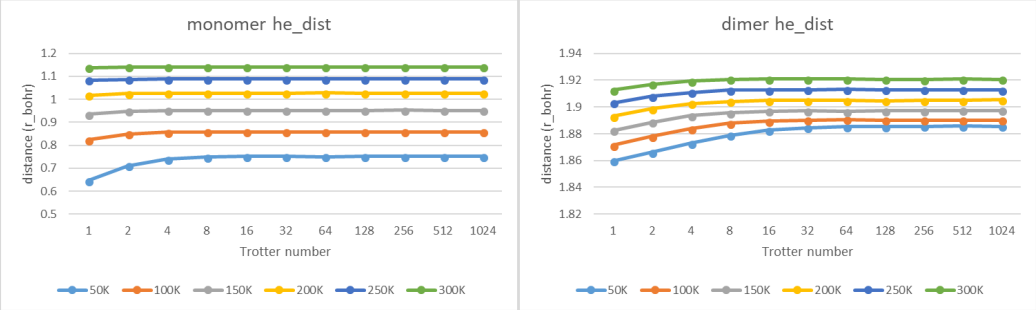

Figure 10: *The mean value of the distance distribution* he_dist *in different SPHERICAL simulations. The estimated SEM:s of these are below* $0.0005 r_{Bohr}$.

The simulation set 1 contains the simulations done with the SPHERICAL model, i.e. the neutral C_{60} spherical cavity model. The statistical reliability of this simulation set is very high, giving approximately statistically independent datapoints even for the highest Trotter number and lowest temperature. If the SEM for any of the time series is considered, it is very small compared to the effects of changing temperature and Trotter number.

The mean distance of the helium atoms from center (the "he_dist" time series) is given in the figure 10. It is seen that increasing temperature increases the distance, which makes sense since the helium atoms can push further into the potential energy wall with higher thermal energy. Also increasing the Trotter number appears to increase the distance until the Trotter number exceeds 32. As should be expected, the lower temperature systems are more quantum mechanical in the sense that higher Trotter number is required to describe them properly.

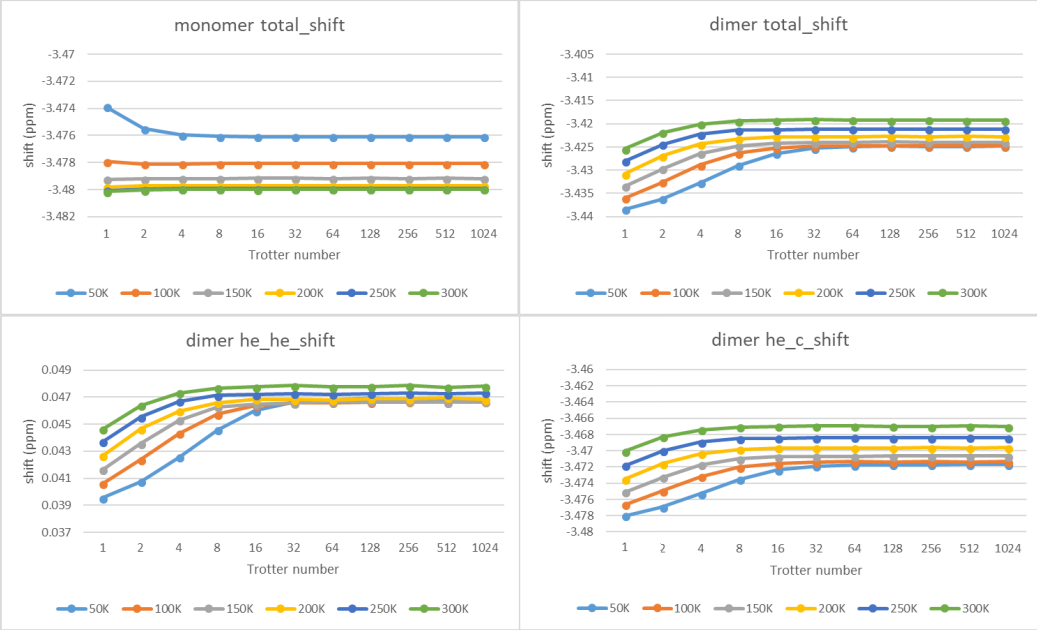

Figure 11: *The mean value of the shift distributions in different SPHERICAL simulations. The estimated SEM:s of these are below* 0.00005ppm.

The mean chemical shifts are given in figure 11. Trotter number of 32 produces again acceptable accuracy. For the monomer the shielding from the carbon atoms (monomer total_shift) is stronger than for the dimer (dimer he_c_shift). This is because of the form of the spherically symmetric NMR force field used, where the shielding remains relatively constant close to the center, but decays fast after the distance from the center is more than about 1.2 Å. In the monomer, the single helium atom remains close to the center, but in the dimer, the two helium atoms push each other agains the walls of the cavity, and to the area where they experience less shielding. Increasing temperature increases shielding slightly for the monomer because the shielding is strongest at a location that is slightly offset from the center. Increasing temperature for the dimer weakens the shielding, which is probably because the helium atoms push further into the walls.

5.2 RIGID model

The simulation set 2 contains the simulations made with RIGID model, i.e. the neutral C_{60} model with rigid fullerene cage. The statistical reliability of the simulation set 2 is also very high. Comparing with the results of simulation set 1, the position averages (figure 12) are very similar, but the shifts

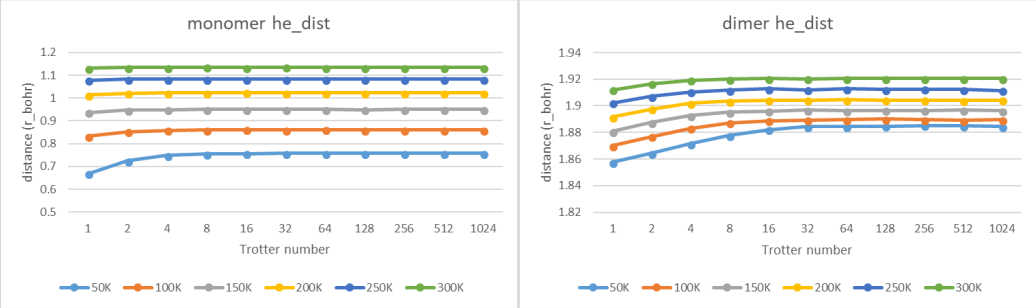

Figure 12: *The mean value of the distance distribution* he_dist *in different RIGID simulations. The estimated SEM:s of these are below* $0.0008 r_{Bohr}$

(figure 13) have slightly changed. Notably the shift vs. temperature trend is more linear in RIGID simulations, about $-4 \cdot 10^{-5}\,\mathrm{ppm/K}$ for monomer and $3 \cdot 10^{-5}\,\mathrm{ppm/K}$ for dimer. Therefore increasing the measurement temperature by about 10 K would increase the shift difference between monomer and dimer by about $7 \cdot 10^{-4}\,\mathrm{ppm} \approx 0.001\,\mathrm{ppm}$. Trotter number of 32 produces good results at all temperatures.

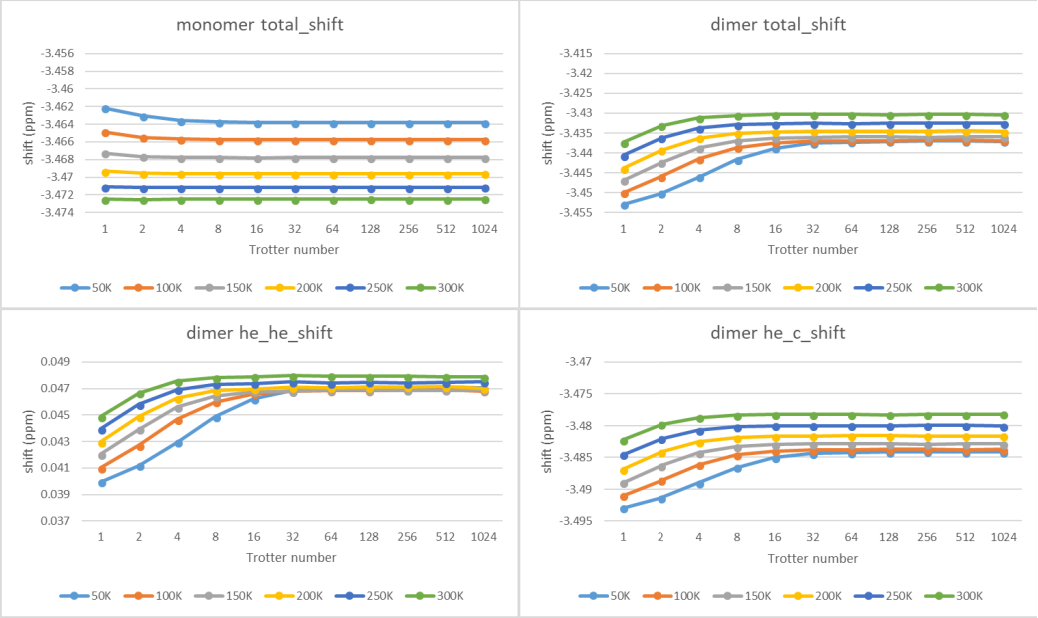

Figure 13: *The mean value of the shift distributions in different RIGID simulations. The estimated SEM:s of these are below* 0.00011ppm.

5.3 FULL model

5.3.1 Statistical considerations

In simulations where the fullerene cage dynamics are simulated, the statistical quality of the results is often low. This is partly due to the fact that there are more degrees of freedom to be simulated, but this is not the only reason. Especially in lower temperatures, the relatively rigid fullerene potential necessitates the use of small trial moves, while the interaction between time slices (that is a loose potential in low temperatures) requires traversal of large distances. This combination of large distances and low traversal speed requires a very large number of trial moves in order to get statistically good results. If one wants to simulate the fullerenes at low temperatures, it is a good idea to include trial moves that rigidly rotate and move all the carbon atoms in one time slice, for example.

In figure 14 the autocorrelation of "c_dist"-time series from the simulation with monomer helium and various Trotter numbers is given. It is seen that the autocorrelations have long tails, which causes errors in the automatic estimation of statistical inefficiency and IAcT. For the $P_{\mathrm{He}} = P_{\mathrm{C}} = 128$ case, the IAcT is about 400, while the time series analysis program ts_ana.x estimated it to be 355. This underestimation is attributed to the cutoff limit.

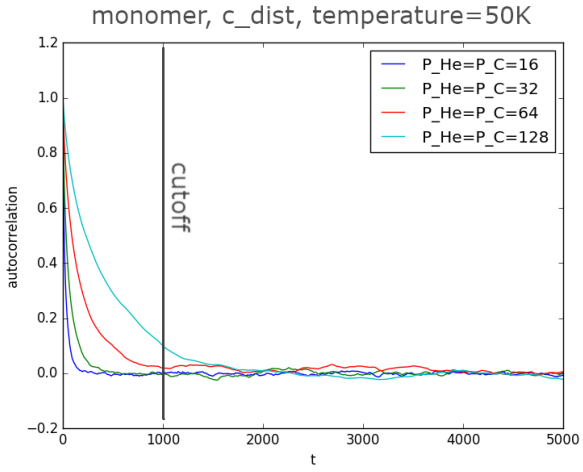

Figure 14: *The autocorrelation function of the "c_dist" time series in various FULL simulations from the simulation set 4.*

The IAcT of 400 means that the correct value for statistical inefficiency would be 800. The program ts_ana.x estimated this value to be 229, which is about 1/4 of the correct value. If the SEM is estimated based on this incorrect value using the eqations (32) and (33), the estimated SEM will be about half of the correct value. A manual estimation is sometimes more accurate than this automatic estimation. However using this automatic analysis system makes sense since manual analysis of the time series would take a long time. Additionally the reliability of the results can be always checked with the sanity check $2\tau_{int} \approx s$, which works best with large simulation sets (so that it is less likely that the sanity condition is fulfilled for all the simulations only because of coincidence).

5.3.2 Position convergence

The average positions are given in figure 15. The estimated SEM:s are largest for low temperature and high Trotter number. It should be noted that the sanity condition concerning statistical inefficiency and IAcT ($2\tau_{int} \approx s$) was not fulfilled for the time series with highest estimated SEM:s, which means that the SEM:s could be underestimated. This was exclusively a problem with the simulations where $P_C \neq 1$. A manual inspection, done for time series with highest estimated SEM, indicated that the SEM should be about twice as high.

It is seen that higher temperature again increases the distance of helium atoms from the center of mass. However, the effect is the opposite for carbon

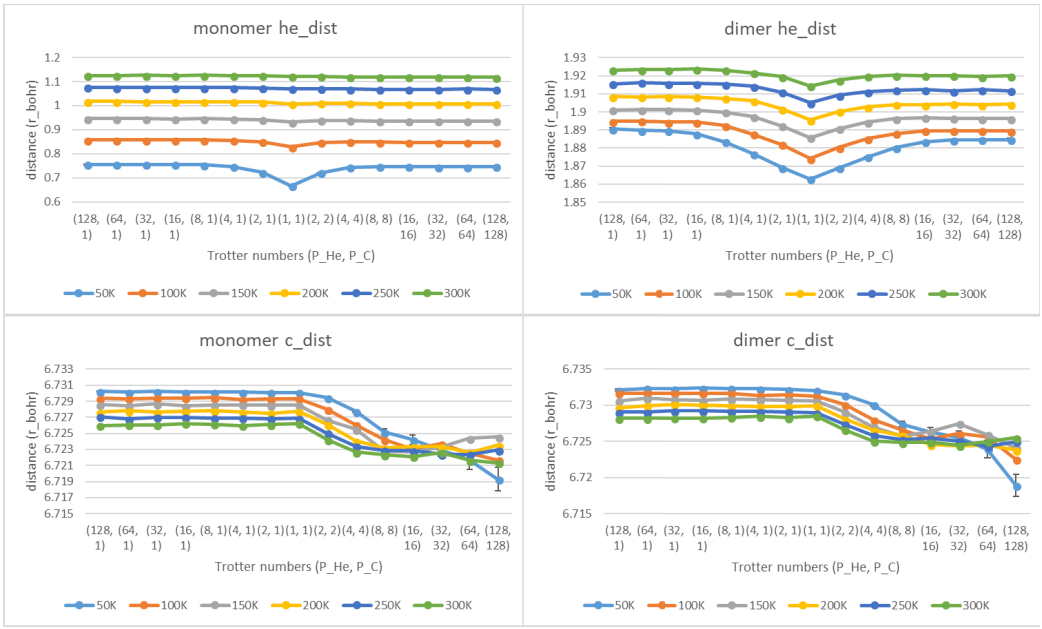

Figure 15: *Mean distances in different FULL simulations. The SEM:s for 50K simulations are plotted as error bars to give an idea of the accuracy of the results. The error bars in other plots are omitted to not make the picture unclear. The plotted error bars are based on automatic estimate and likely underestimated.*

atoms; the fullerene cage compresses when heated. Perhaps the used fullerene potential is "softer" towards center, meaning that the energy required to compress the fullerene molecule is smaller than the energy required to expand it by a similar amount. Further investigation would be required to confirm this.

The effect of Trotter numbers to the positions is such that the helium position mainly depends only on P_{He} but is also slightly dependent (especially dimer) on P_{C}. The distance of carbon atoms from center of mass does not depend on P_{He}, but does depend on P_{C}, which should be expected.

Increasing P_{C} compresses the fullerene molecule. This could again be because of the fullerene potential being softer towards the center, since the quantum mechanical effects allow the beads to explore areas of higher potential energy further. This could also be because of relative rotation of the fullerene cages of different time slices, which should on average cause an effective invard force to the carbon atoms. Also this could be because of the combined effect of the relative motion between the cages and the fullerene potential that is "softer" towards centre.

5.3.3 Chemical shift

Figure 16: *The mean value of different shift distributions in different FULL simulations. The estimated SEM:s for these are below* 0.0003 ppm*, but sanity checking indicates again that this is underestimated for some simulations, specifically when* $T < 200K$ *and* $P_C \neq 1$*.*

The chemical shifts estimated from FULL simulations are given in figure (16). There is again some problem when estimating the autocorrelation in the data. The "correct" autocorrelation was not manually inspected, but it is reasonable to assume that the error is underestimated in simulations where the sanity check does not hold. Accordingly the simulations where $P_{\mathrm{C}} \neq 1$ should not be trusted too much. However, it would seem that the results are consistent enough to show that both P_{He} and P_{C} have an effect on the shifts. The temperature dependence of total shift is again in line with the other simulations; the He total shift temperature gradient is approximately for monomer $-4 \cdot 10^{-5}$ ppm/K and for dimer $2 \cdot 10^{-5}$ ppm/K.

5.4 Comparison of the models SPHERICAL, RIGID and FULL

The results obtained so far, i.e. the results that were discussed in the previous chapters, are statistically very good, with some exceptions. Specifically, for the FULL simulation type, obtaining statistically good results appeared to be

difficult at low temperatures when $P_C \neq 1$, where especially the convergence of carbon atom positions proved difficult. Therefore the simulation time needs to be higher when similar simulations are run.

It was seen that, according to the simulations, the observables (mean values of shifts or values calculated from position vectors) have various temperature dependencies, which often are almost linear in the simulated temperature range.

Importantly, it was seen that the observables appear to converge to some value as the Trotter number is increased, which should be expected for a succesful simulation set. From the graphics presented in the previous chapters and in appendix A, it is seen that Trotter number of 32 produces good results at the selected temperature range. The highest Trotter number is of course required for lowest temperature, so at higher temperatures, lower Trotter number could also be used. Also it may be that a smaller Trotter number would be enough for carbon atoms since they are heavier, and the bead-bead harmonic force is stronger for heavier atoms. On the other hand, the carbon atoms are bound more rigidly compared to the helium atoms. This would imply that using a high Trotter number would be required also for the carbon atoms, since even small changes in the path may translate into high energy differences. The equation $P >> \beta\hbar^2/(m\sigma^2)$, where P, β, m, σ are the Trotter number, the inverse temperature, mass and length scale of the potential, respectively, describes approximately the required Trotter number[19]. Therefore reducing the length scale of potential by a factor of two would require increasing the Trotter number by a factor of four.

At temperature 300 K the results can be summarized to figure 17, which gives the total shift in various simulations. All these simulations produced somewhat similar results, with the shift difference between monomer and dimer ranging from -0.035 to -0.061 ppm. Unfortunately this can not be compared to a measured result because the dimer peak is hidden under the monomer peak in the spectrum[4]. Both the used simulation type and level of quantum mechanics (Trotter numbers) affected the results.

As seen from the figure 17, for monomer there is no difference between the classical and half-QM models. This is because in monomer the helium atom experiences a relatively loose external potential, and therefore the quantum mechanical description does not change the position distribution much. Additionally, a small change in the position distribution does not change the shift unless it can push the helium atom close enough to the walls of the cavity, where it experiences weaker shielding. For the dimer, the situation is different. In dimer, the helium atoms are in a more tighter potential well, and therefore the classical model falls short in predicting the position distribution. Additionally, as the helium atoms are already close to the walls

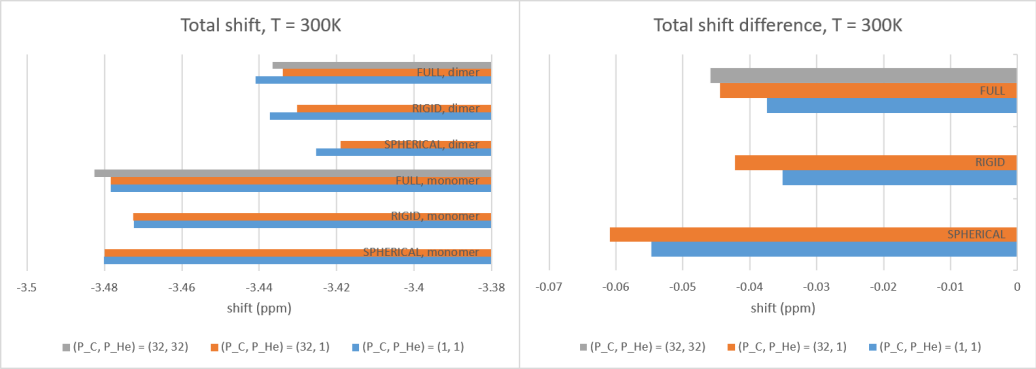

Figure 17: *Summary of total shifts in SPHERICAL, RIGID and FULL models at $T = 300\,\mathrm{K}$. Three different quantum mechanical approximations; classical ($P_{He} = P_C = 1$), half quantum mechanical ($P_{He} = 32, P_C = 1$) and full quantum mechanical ($P_{He} = P_C = 32$). The error (SEM) for these is insignificant ($< 0.00009ppm$).*

of the cavity, only a small change in the position distribution will allow the helium atom to escape to the area of weaker shielding. Indeed, as seen from the figure 17, the half-QM description decreases the shielding.

Figure 17 also shows that in the FULL model the quantum mechanics of the carbon atoms increases shielding for both dimer and monomer. From figure 16 we can see that this is due to the shielding from the fullerene for both dimer and monomer.

5.5 NEW models

This chapter summarizes the results of the simulation set 5, which comprises two improved models; the neutral C_{60} model and the anion C_{60}^{6-} model. These models were simulated at three different quantum mechanical approximations; classical ($P_{\mathrm{He}} = P_{\mathrm{C}} = 1$), half quantum mechanical ($P_{\mathrm{He}} = 32, P_{\mathrm{C}} = 1$) and full quantum mechanical ($P_{\mathrm{He}} = P_{\mathrm{C}} = 32$). The choice of Trotter numbers were made according to what was detected to be good enough in the previous simulations. Although lower Trotter numbers could have been selected for higher temperatures, I chose to use the same Trotter numbers at all temperatures for simplicity.

Since the simulation time of 10 hours appeared to not give good results for all situations, the simulation time for this simulation set was increased to 100 hours. Although lower simulation time would have been enough for many cases (e.g. the classical case), the same simulation time was specified for simplicity. The simulation time of 100 hours was enough to give statistically

good results for all the simulated cases, and the time series analysis program had no problems in estimating the statistical reliability.

The results of this simulation set can be summarized to four pictures (18, 19, 20 and 21), which depict the mean values of different time series. In each subpicture, the maximum error (SEM) within that subpicture is depicted as the error bars of the 50 K plot (the error bars are not visible if error is small). Therefore, for each subpicture, only one error value is depicted, which gives the largest SEM. The SEM:s for the classical simulations are of course often smaller than for half-QM or full-QM simulations, but since the SEM is so small for all time series depicted, the maximum SEM is only depicted for simplicity.

Both of these new models confirm the assessment of the previous chapter that the carbon atom distance from center decreases when P_{C} increases. The mean distances change in a similar way with increasing temperatures as in the FULL model; the helium atoms move away from the center, and the carbon atoms move towards center.

In both of these new models, it is confirmed that temperature and both Trotter numbers have an effect on the chemical shifts. Temperature induced change in the shifts seem to be much stronger in the anion model, where the He-C-shift varies in the range of 0.1 ppm with varying temperatures in the 50K - 300K range. In the neutral model, the changes in shifts are only around 0.01 ppm in the same temperature range. Since the distances change in a similar way both in anion and neutral models, this sensitivity of the anion shift is simply due to sensitivity of the NMR force field used to describe the shielding of the fullerene cage. Depending on the used QM-level, increasing temperature may either increase or decrease the shielding for anion dimer.

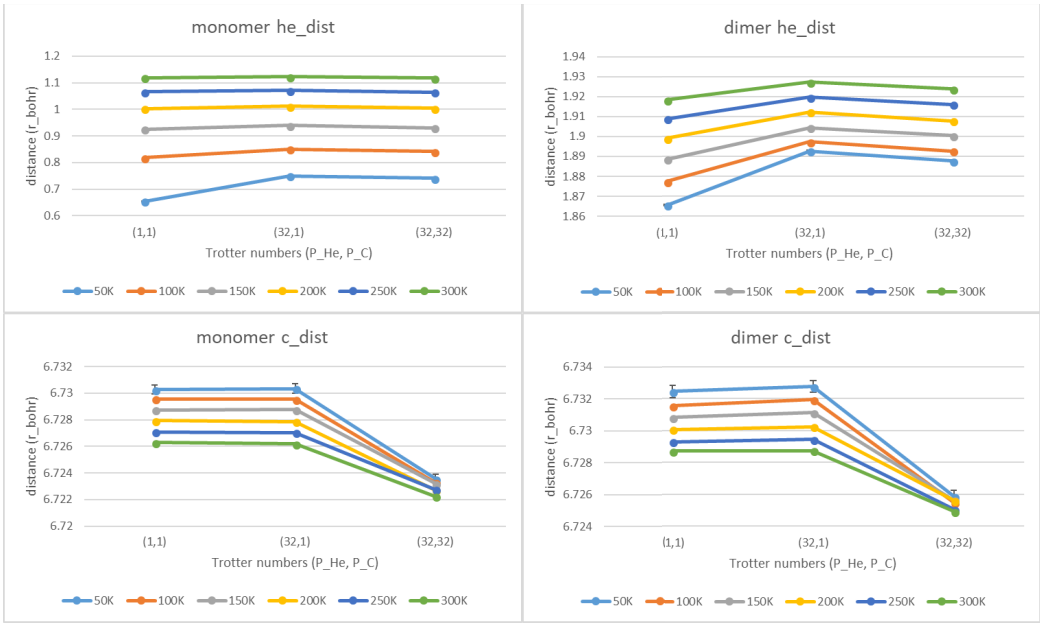

Figure 18: *The distances in the neutral model (FULL_NEW_NEUTRAL).*

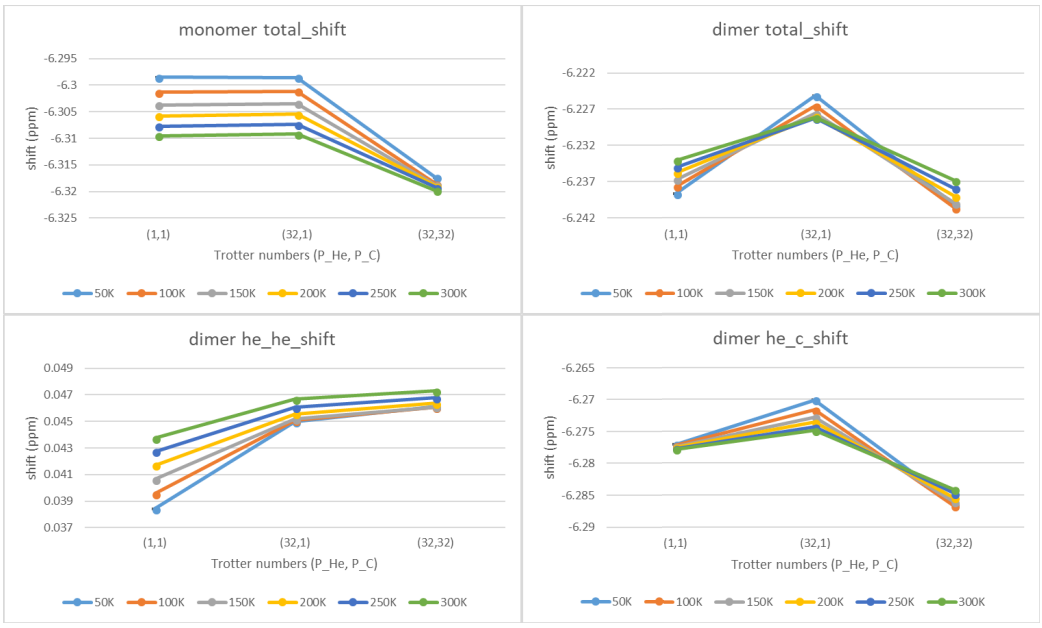

Figure 19: *The shifts in the neutral model (FULL_NEW_NEUTRAL).*

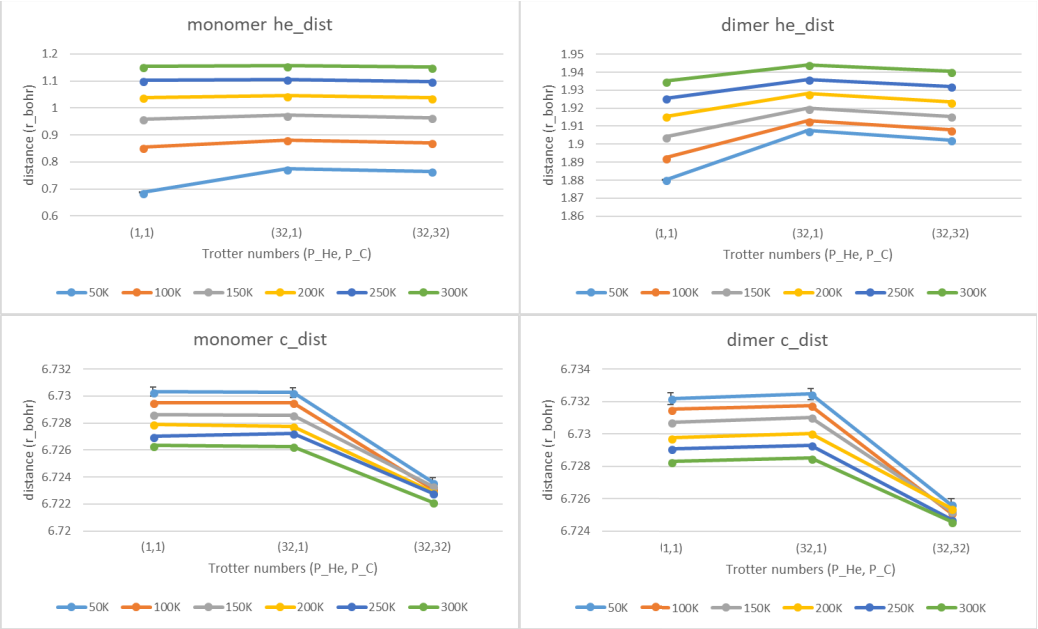

Figure 20: *The distances in the anion model (FULL_NEW_ANION).*

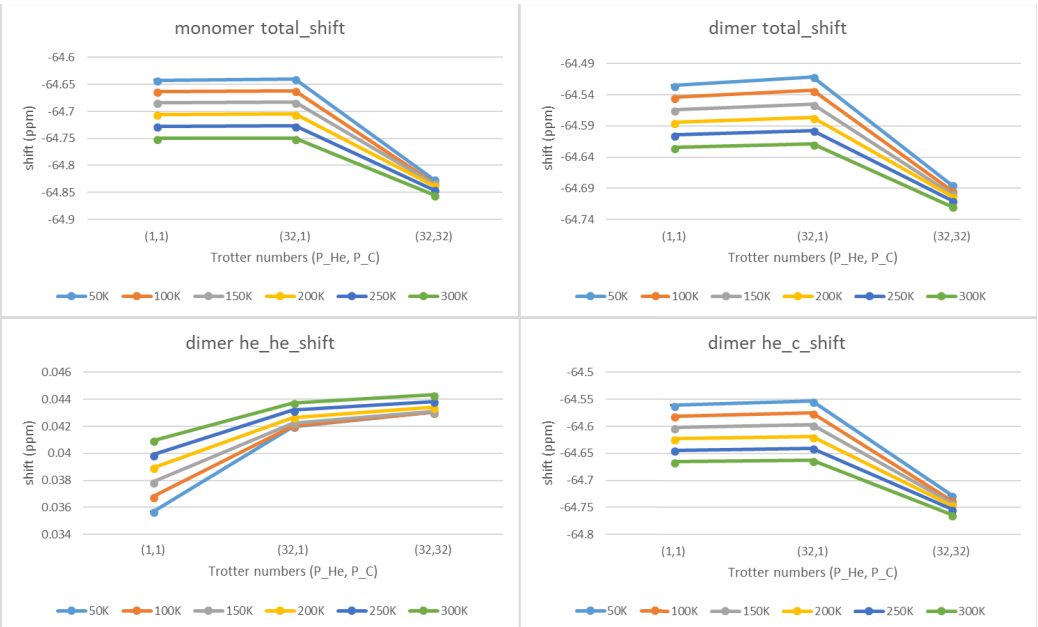

Figure 21: *The shifts in the anion model (FULL_NEW_ANION).*

In the table 15 is given the shifts estimated based on the simulation set 5 in temperature 300 K. The simulated value for the shift difference of monomer and dimer is approximately -0.08 ppm for neutral fullerene and -0.13 ppm for anion. The anion shift difference can be compared with the experimental result[4] of -0.093, which is quite close.

The figure 22 shows the shift difference between total shifts of monomer and anion.

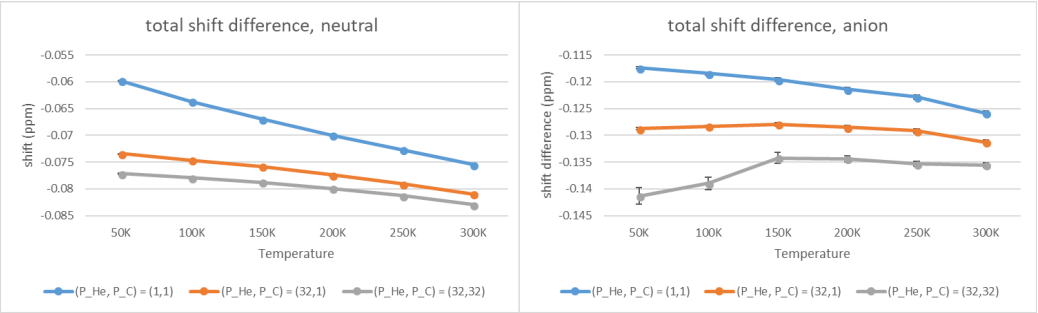

Figure 22: *Shift differences between dimer and monomer*

Table 15: *Summary of 300K simulations in simulation set 5. The reported error values are the SEM:s. The shift difference SEM has been calculated from the two SEM:s as* $\sqrt{SEM_1^2 + SEM_2^2}$. *The experimental "Anion experimental" row is the result from [4].*

	dimer			monomer	dimer/monomer
	He-He-shift (ppm)	He-C-shift (ppm)	total shift (ppm)	total shift (ppm)	total shift difference (ppm)
Neutral classical	0.04373 ± 0.00003	−6.27780 ± 0.00004	−6.23407 ± 0.00004	−6.30952 ± 0.00003	−0.07545 ± 0.00005
Neutral half-QM	0.04669 ± 0.00004	−6.27487 ± 0.00004	−6.22818 ± 0.00005	−6.30916 ± 0.00003	−0.08099 ± 0.00006
Neutral QM	0.0473 ± 0.00004	−6.28416 ± 0.00005	−6.23686 ± 0.00006	−6.31982 ± 0.00004	−0.08296 ± 0.00007
Anion classical	0.04096 ± 0.00003	−64.6655 ± 0.0003	−64.6245 ± 0.0003	−64.7503 ± 0.0003	−0.1258 ± 0.0005
Anion half-QM	0.04374 ± 0.00004	−64.6626 ± 0.0003	−64.6188 ± 0.0003	−64.7501 ± 0.0003	−0.1313 ± 0.0004
Anion QM	0.04433 ± 0.00004	−64.7635 ± 0.0004	−64.7191 ± 0.0004	−64.8547 ± 0.0004	−0.1356 ± 0.0005
Anion experimental					−0.093

6 Conclusion

In this thesis multiple physical models of the endohedral fullerenes $He_1@C_{60}$, $He_2@C_{60}$, $He_1@C_{60}^{6-}$ and $He_2@C_{60}^{6-}$ were simulated using Monte Carlo methods. The position distributions of atoms and chemical shifts were observed, and the simulation time was increased until the statistical quality of the observed averages could be verified.

The observed quantities showed various dependencies from Trotter numbers and temperature. After increasing the Trotter number above certain limit, the observed quantities seemed to converge. Based on this, at T=50K the required Trotter number was estimated to be around 32. At higher temperatures lower Trotter numbers are enough. The chemical shift difference between endohedral fullerenes $He_1@C_{60}^{6-}$ and $He_2@C_{60}^{6-}$ was especially interesting, since it could be compared with experimental results. Experimentally this shift difference has been obserbed to be $-0.093\,\mathrm{ppm}$[4], while the anion model simulated with $P_{\mathrm{He}} = P_{\mathrm{C}} = 32$ at $T = 300\,\mathrm{K}$ produced the result $-0.1356 \pm 0.0005\,\mathrm{ppm}$, which is relatively close.

The difference between the experimental and computational result could be due to many reasons. One limitation of the simulations was that the thermalization phase of the simulations was partly skipped, i.e. the equilibration of the system energy was not verified before starting the datacollection phase. This could cause error in the results if the system did not have enough time to reach equilibrium. However, due to the simulation being run for a relatively long time during the trial move optimization phase, I think that the system is likely to be well equilibrated after the optimization. It is possible that the used physical model is not accurate enough. This could happen if the system

is not accurately described by Boltzmannian statistics, or the approximation caused by the effective potential and NMR force fields.

Possible future directions for research would be to improve the current models of C_{60} and make a model of the C_{70} fullerene, for which experimental results also exist[4]. For improving the current models, it would need to be determined whether the error is due to the force field approximation, NMR force field approximation or some other source. The C_{70} model could possibly use the same pairwise interactions as the C_{60} models. However, a different force field would be needed for describing the dynamics of the C_{70} fullerene cage. One possible choice would be to use the general forcefields defined by *Program Fullerene*[39].

7 Acknowledgements

Some of the imagery in this thesis was produced using the matplotlib[40] library. The simulations were done in the Carpo computer cluster which is a part of the Finnish Grid and Cloud Infrastructure project (FGCI) (persistent identifier: urn:nbn:fi:research-infras-2016072533).

References

[1] Arnout Ceulemans, Bruno C. Titeca, Liviu F. Chibotaru, Ingrid Vos, and Patrick Fowler. Complete bond force fields for trivalent and deltahedral cages: Group theory and applications to cubane, closo-dodecaborane, and buckminsterfullerene. *J. Phys. Chem. A*, 105:8284–8295, 09 2001.

[2] P. Loftus R. J. Abraham, J. Fisher. *Introduction to NMR Spectroscopy.* John Wiley and Sons, 1988.

[3] Teobald Kupka, Michał Stachów, Leszek Stobiński, and Jakub Kaminský. 3He NMR: from free gas to its encapsulation in fullerene. *Magnetic Resonance in Chemistry*, 51(8):463–468, 2013.

[4] Tamar Sternfeld, Roy E. Hoffman, Martin Saunders, R. James Cross, M. S. Syamala, and Mordecai Rabinovitz. Two helium atoms inside fullerenes: Probing the Internal Magnetic Field in C_{60}^{6-} and C_{70}^{6-}. *Journal of the American Chemical Society*, 124(30):8786–8787, Jul 2002.

[5] Katja Hyvönen. Heliumatomin ja -dimeerin nmr kemialliset siirtymät C_{60}-fullereenin muodostamassa keskeispotentiaalissa polkuintegraali-Monte Carlo menetelmällä, 2015. Master's thesis.

[6] Tsutomu Kawatsu and Masanori Tachikawa. The quantum fluctuations of the fullerene cage modulate the internal magnetic environment. *Physical Chemistry Chemical Physics*, 20, 12 2017.

[7] R. P. Feynman. Space-time approach to non-relativistic quantum mechanics. *Rev. Mod. Phys.*, 20:367–387, Apr 1948.

[8] Markku Leino. Finite-temperature quantum statistics of a few confined electrons and atoms - path-integral approach, 2007.

[9] Burkhard Militzer. *Path Integral Monte Carlo Simulations of Hot Dense Hydrogen.* PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 2000.

[10] Yimin Li and William H. Miller *. Different time slices for different degrees of freedom in feynman path integration. *Molecular Physics*, 103(2-3):203–208, 2005.

[11] K. Blum. *Density Matrix Theory and Applications, Second Edition.* Plenum Press, New York, 1996.

[12] David J. Griffiths. *Introduction to Quantum Mechanics, second edition.* Pearson Prentice Hall, 2005.

[13] M. J. Gillan. *The Path-Integral simulation of quantum systems.* Kluwer Academic Publishers, 1990. chapter in book: Computer Modeling of Fluids Polymers and Solids, editors C. R. A. Catlow, S. C. Parker, M. P. Allen.

[14] W. L. Dunn. *Exploring Monte Carlo Methods.* Academic Press, 2012.

[15] Andrei Andreevich Markov. Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga. *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete*, 15(135-156):18, 1906.

[16] Serge Bernstein. Sur l'extension du théoréme limite du calcul des probabilités aux sommes de quantités dépendantes. *Mathematische Annalen*, 97(1):1–59, Dec 1927.

[17] Malvin H. Kalos and Whitlock Paula A. *Monte Carlo Methods.* Wiley, 2008. ISBN: 978-3-527-40760-6.

[18] Niclolas Metropolis, Arianna W. Rosenbluth, Rosenbluth Marshall N., Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21:1087–1092, 1953.

[19] Llorenç Brualla i Barberà. Path integral monte carlo algorithms and applications to quantum fluids, 2002. Doctoral Dissertation.

[20] M. B. Priestley. *Spectral Analysis and Time Series*. Academic Press, 1981. ISBN: 0-12-564950-9.

[21] Robert D. Skeel and Youhan Fang. Comparing markov chain samplers for molecular simulation. *Entropy*, 19(10), 2017.

[22] Kari Rummukainen. Monte carlo simulations in physics. (Lecture notes from course: Monte Carlo simulation methods, available from course website: `http://www.helsinki.fi/~rummukai/lectures/montecarlo_oulu/`.

[23] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 1989.

[24] Marcin Janicki, Banaszczyk Jedrzej, Bjorn Vermeersch, Gilbert De Mey, and Andrzej Napieralski. Generation of reduced dynamic thermal models of electronic systems from time constant spectra of transient temperature responses. *Microelectronics Reliability*, 51:1351–1355, 2011.

[25] Elke Pahl, Detlev Figgen, Christian Thierfelder, Kirk Peterson, Florent Calvo, and Peter Schwerdtfeger. A highly accurate potential energy curve for the mercury dimer. *The Journal of chemical physics*, 132:114301, 03 2010.

[26] Kenneth Hedberg, Lise Hedberg, Donald S. Bethune, C A Brown, Harry Dorn, Robert Johnson, and Mattanjah de Vries. Bond lengths in free molecules of buckminsterfullerene, c60, from gas-phase electron diffraction. *Science (New York, N.Y.)*, 254:410–2, 11 1991.

[27] Markku Alamäki, Petr Štěpánek, Juha Vaara, Michal Straka, and Perttu Lantto. Dynamical contribution to the chemical shift of helium atoms in fullerene cage studied by path integral monte-carlo. Manuscript of unpublished work.

[28] David E. Woon and Thom H. Dunning Jr. Gaussian basis sets for use in correlated molecular calculations. iv. calculation of static electrical response properties. *The Journal of Chemical Physics*, 100:2975, 1994.

[29] Florian Weigend, Marco Häser, Holger Patzelt, and Reinhart Ahlrichs. Ri-mp2: optimized auxiliary basis sets and demonstration of efficiency. *Chemical Physics Letters*, 294(1-3):143–152, 1998.

[30] Andreas Hesselmann and Tatiana Korona. On the accuracy of dft-sapt, mp2, scs-mp2, mp2c, and dft+disp methods for the interaction energies of endohedral complexes of the c60 fullerene with a rare gas atom. *Physical Chemistry Chemical Physics*, 13:732–743, 2011.

[31] Dalton, a molecular electronic structure program, release dalton2017 (2017), see http://daltonprogram.org.

[32] A. D. Becke. Density-functional exchange-energy approximation with correct asymptotic behavior. *Physical Review A*, 38:3098–3100, 1988.

[33] Chengteh Lee, Weitao Yang, and Robert G. Parr. Development of the colle-salvetti correlation-energy formula into a functional of the electron density. *Physical Review B*, 37:785–789, 1988.

[34] Axel D. Becke. A new mixing of hartree-fock and local density-functional theories. *The Journal of Chemical Physics*, 98:1372–1377, 1993.

[35] Frank Jensen. Segmented contracted basis sets optimized for nuclear magnetic shielding. *Journal of Chemical Theory and Computation*, 11(1):132–138, 2015.

[36] website: `https://gcc.gnu.org/onlinedocs/gfortran/RANDOM_005fNUMBER.html`, accessed 12.5.2018.

[37] JsonCpp, a C++ library, available from GitHub: `https://github.com/open-source-parsers/jsoncpp`.

[38] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[39] Peter Schwerdtfeger, Lukas Wirz, and James Avery. Program fullerene: A software package for constructing and analyzing structures of regular fullerenes. *Journal of computational chemistry*, 06 2013.

[40] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

A Some histograms

This section contains histograms generated from various time series. The purpose of this section is to demonstrate the convergence of these distributions with respect to the Trotter number. First the figures (23-27) depict convergence of the FULL model, which include distribution plots of all the dimer time series, which are the "c_dist", "he_dist", "total_shift", "he_he_shift" and "he_c_shift". The monomer distributions are omitted since they are very similar and add no significant additional information. Also only the case of equal Trotter numbers for helium and carbon is considered. From the figures (23-27) it is seen that the Trotter number of 32 seems to be enough to converge the distributions. However from only these figures it can not be yet determined wether the helium Trotter number or the carbon Trotter number is more important. This does not matter though, since we only want to show that selected Trotter number of 32 will be good enough to converge the distributions of interest.

Next figures (28-38) show the distributions of the newer simulations (the models "FULL_NEW_NEUTRAL" and "FULL_NEW_ANION"). Again the same five time series are considered, with some of the plots omitted if they are very similar to others. For example the carbon position distribution is almost same for all four cases ($He_1@C_{60}$, $He_2@C_{60}$, $He_1@C_{60}^{6-}$, and $He_2@C_{60}^{6-}$), so only one case is plotted here. Also for monomer case, the total shift and helium-carbon shift are exactly same so only one of them is plotted. The table 16 defines which distributions are omitted.

Table 16: *The included distributions. The carbon distribution c_dist should be similar for all cases, so only one of them is included. The helium atom distribution is similar for anion and neutral cases but different for monomer and dimer cases. The he_c_shift is omitted for monomer because it is the same as total_shift.*

	neutral monomer	neutral dimer	anion monomer	anion dimer
c_dist	yes	no (similar)	no (similar)	no (similar)
he_dist	yes	yes	no (similar)	no (similar)
total_shift	yes	yes	yes	yes
he_he_shift	no (N/A)	yes	no (N/A)	yes
he_c_shift	no (similar)	yes	no (similar)	yes

Looking at the figures (28-38) it is seen unsurprisingly that the carbon position distribution is sensitive to carbon Trotter number. A similar observation is made about helium position distribution and Trotter number. The inclusion of quantum mechanical effects through the Trotter number makes the corresponding position distributions wider. From this, it makes sense to

assume that the shift from interaction between helium atoms (he_he_shift), which depend only on the position of helium atoms, is only affected by the changes in helium Trotter numbers. This assumption seems to be true. On the other hand, since he_c_shift depends on both the location of carbon and helium atoms, it would make sense that the he_c_shift distribution would depend on both the helium and carbon Trotter numbers. This indeed seems to be true, and therefore both Trotter numbers are important in predicting the shifts.

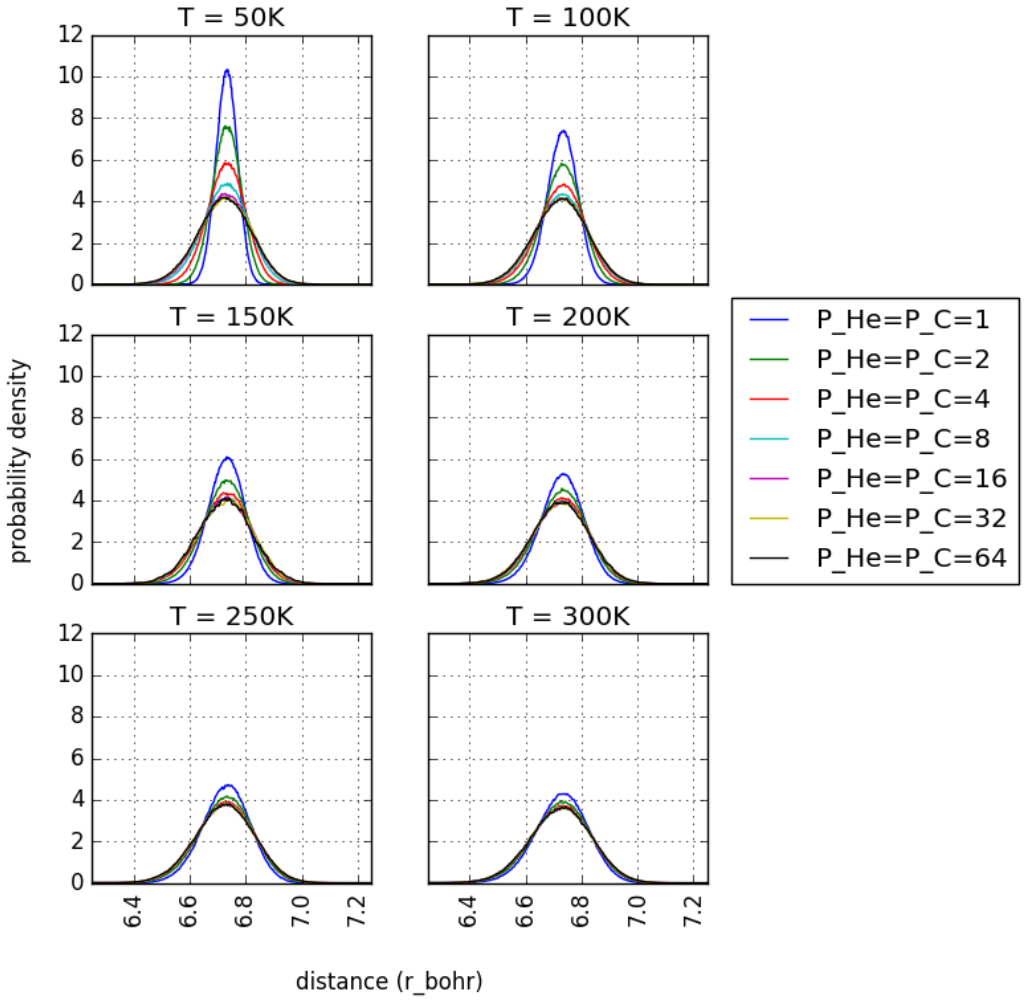

Figure 23: *Neutral dimer ($He_2@C_{60}$) carbon atom position (c_dist) distribution according to the "FULL" model.*

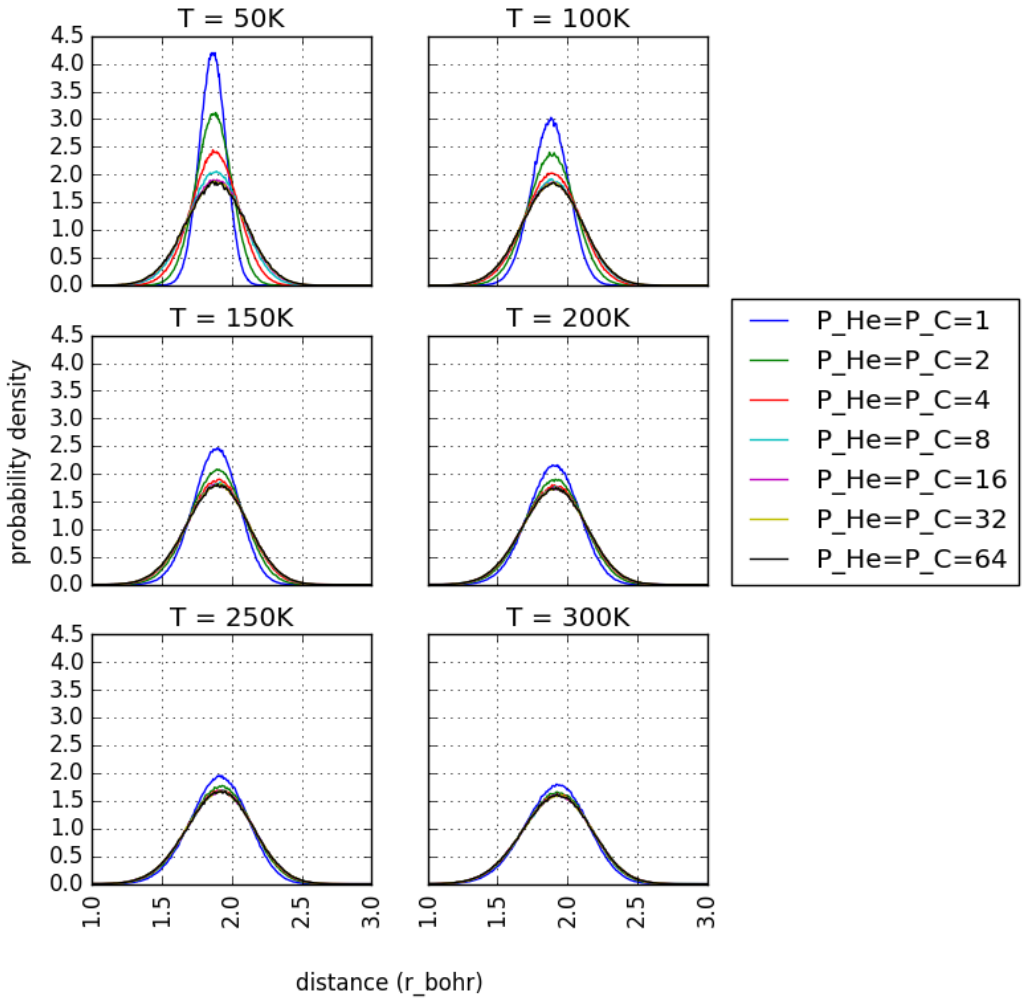

Figure 24: *Neutral dimer (He_2@C_{60}) helium atom position (he_dist) distribution according to the "FULL" model.*

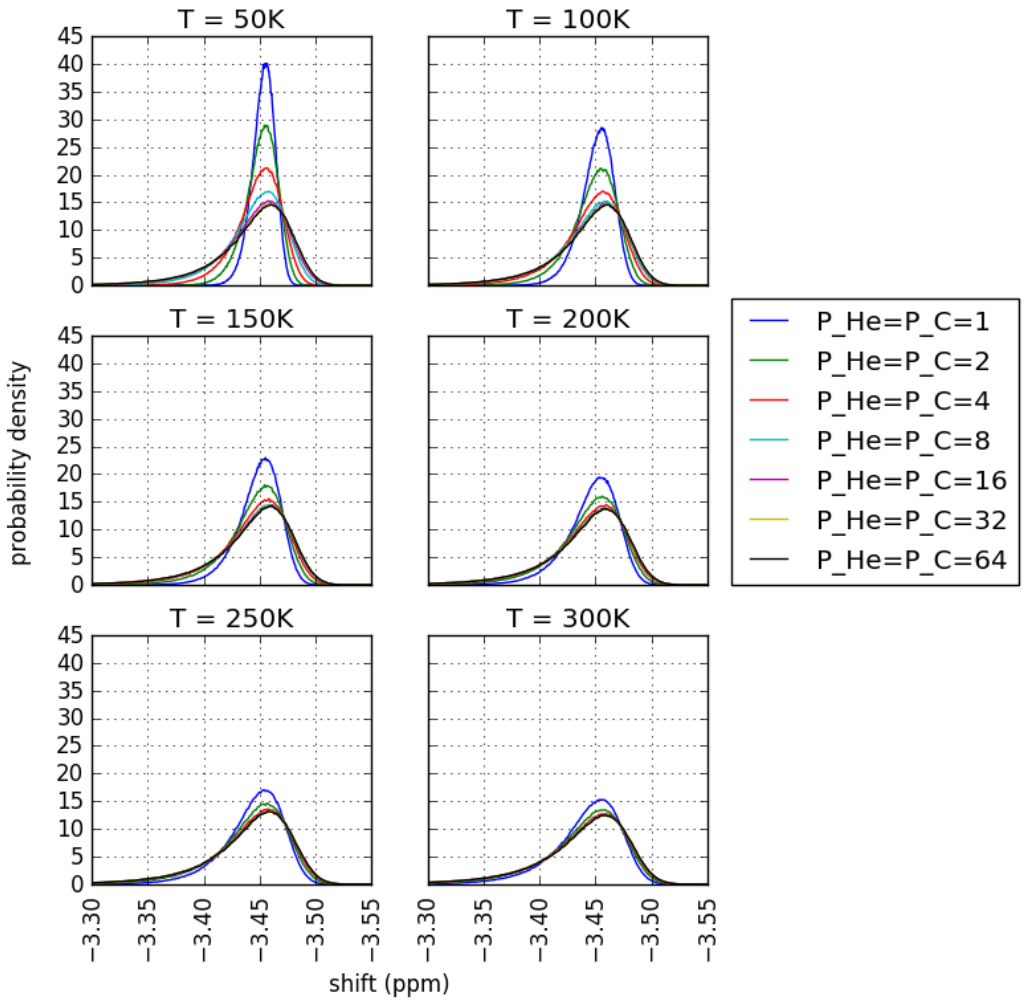

Figure 25: *Neutral dimer ($He_2@C_{60}$) total shift (total_shift) distribution according to the "FULL" model.*

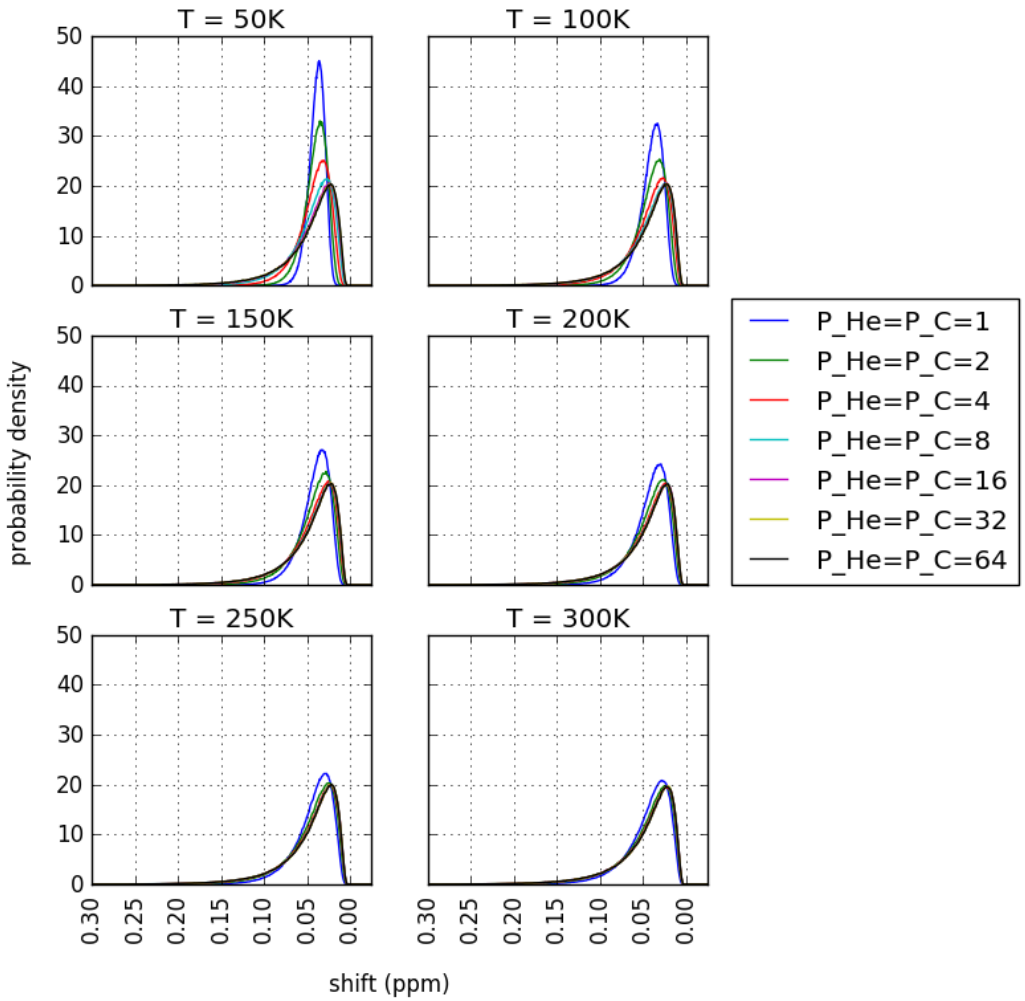

Figure 26: *Neutral dimer ($He_2@C_{60}$) helium-helium shift (he_he_shift) distribution according to the "FULL" model. The quantum mechanical effect allows the two helium atoms to get closer to each other where they experience strong negative shielding (positive shift), which causes the shift distribution to spread here.*

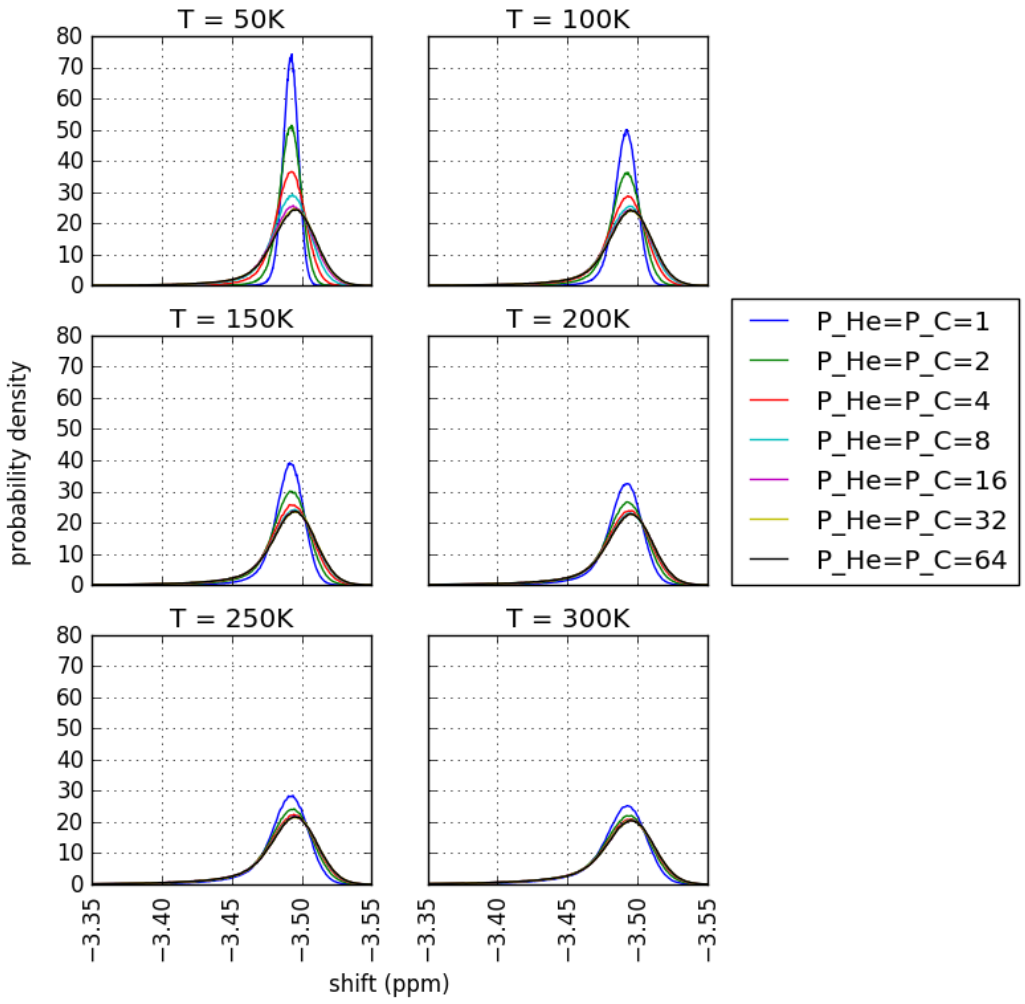

Figure 27: *Neutral dimer ($He_2@C_{60}$) helium-carbon shift (he_c_shift) distribution according to the "FULL" model.*

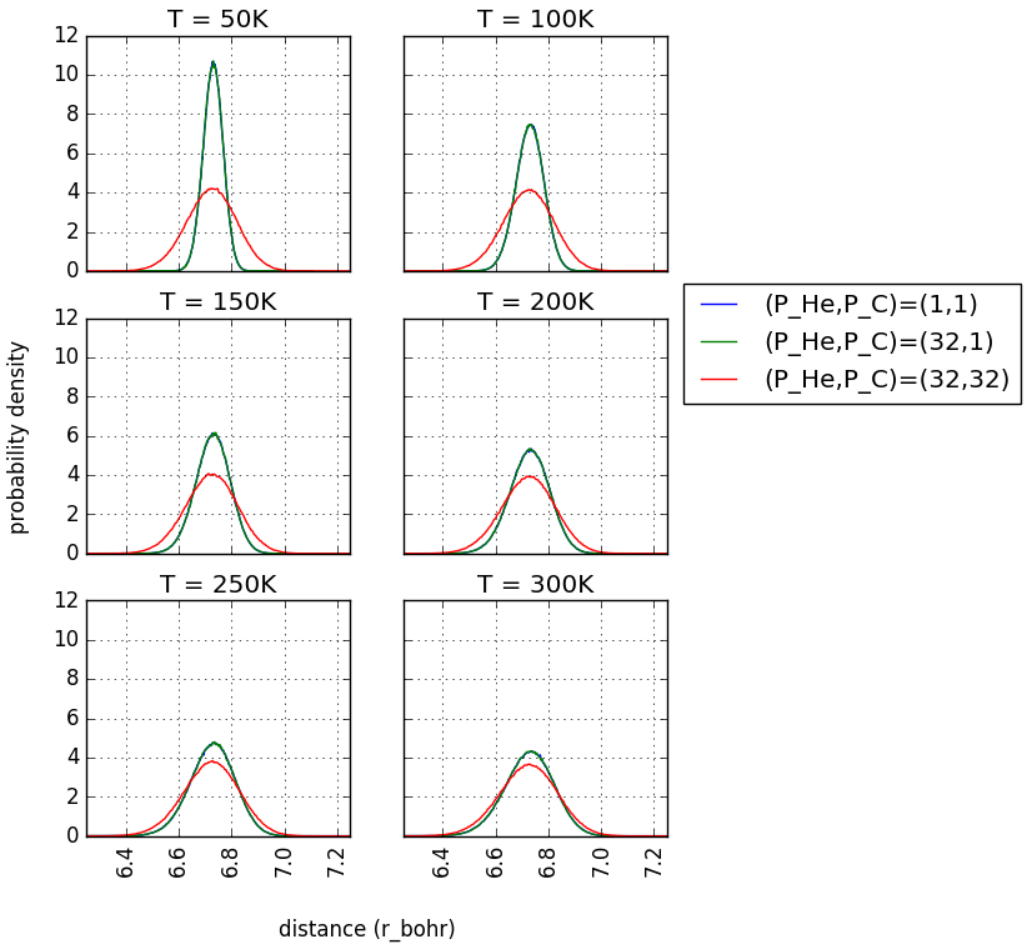

Figure 28: *Neutral monomer ($He_1@C_{60}$) carbon atom position (c_dist) distribution according to the "FULL_NEW_NEUTRAL" model. The $(P_{He}, P_C) = (1, 1)$ and $(P_{He}, P_C) = (32, 1)$ curves are on top of each other because the carbon atom distribution is not much affected by P_{He}.*

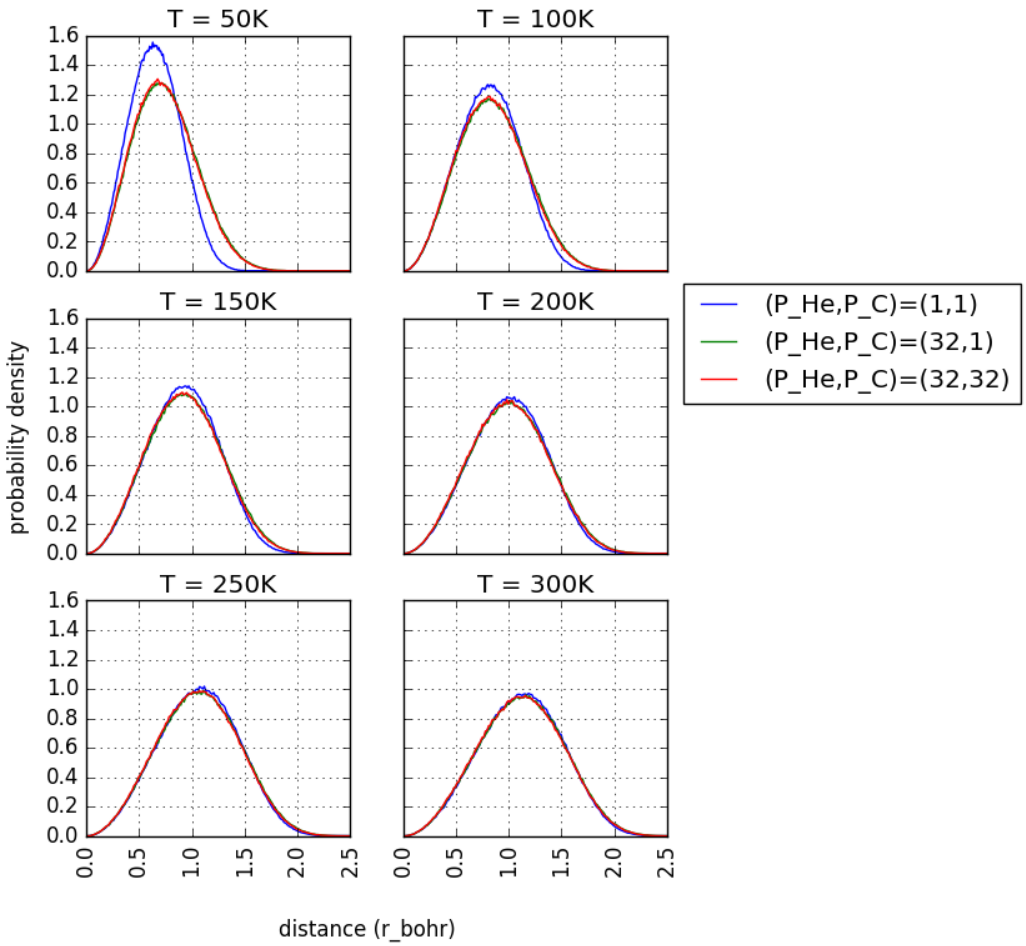

Figure 29: *Neutral monomer ($He_1@C_{60}$) helium atom position (he_dist) distribution according to the "FULL_NEW_NEUTRAL" model. The $(P_{He}, P_C) = (32, 1)$ and $(P_{He}, P_C) = (32, 32)$ curves are on top of each other because the helium atom distribution is not much affected by P_C.*

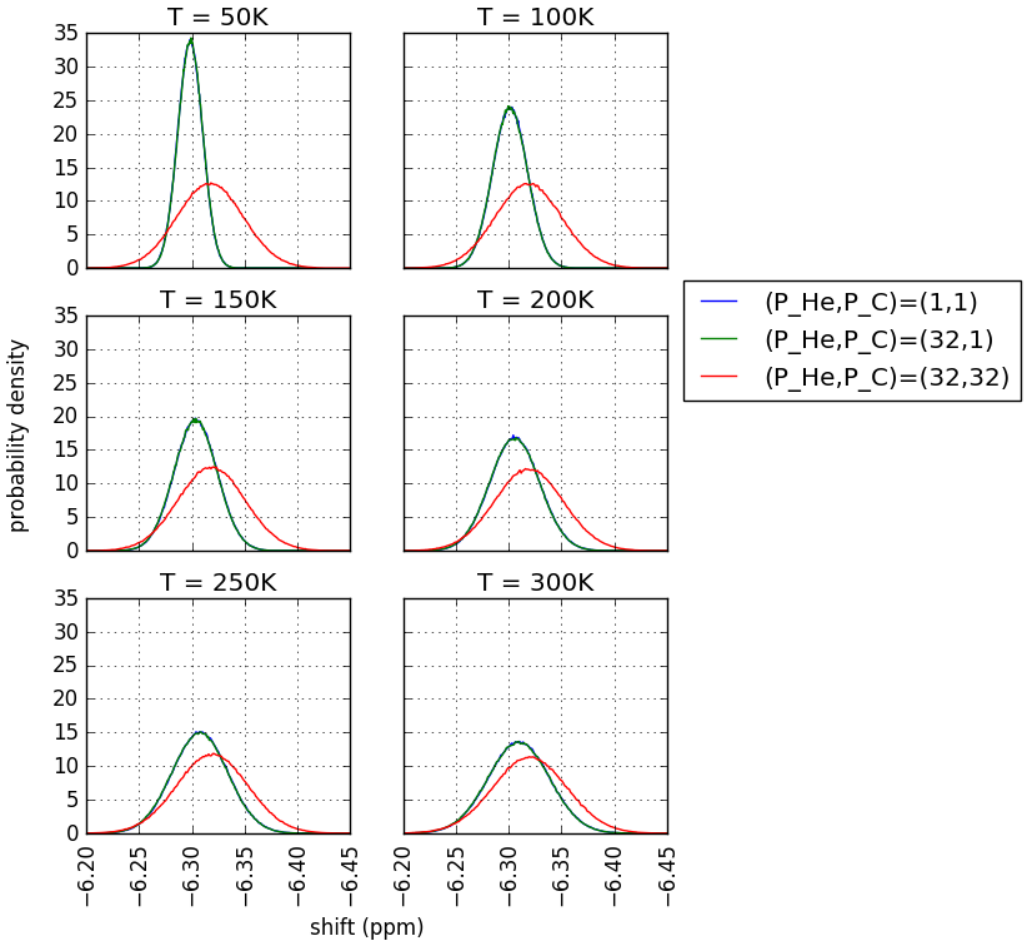

Figure 30: *Neutral monomer ($He_1@C_{60}$) total shift (total_shift) distribution according to the "FULL_NEW_NEUTRAL" model. The $(P_{He}, P_C) = (1, 1)$ and $(P_{He}, P_C) = (32, 1)$ curves are on top of each other, i.e. only P_C makes a difference here. Since the helium position distribution only depended on P_{He} and carbon distribution depended on only P_C, it seems that the shielding depends directly on the carbon atom position distribution.*

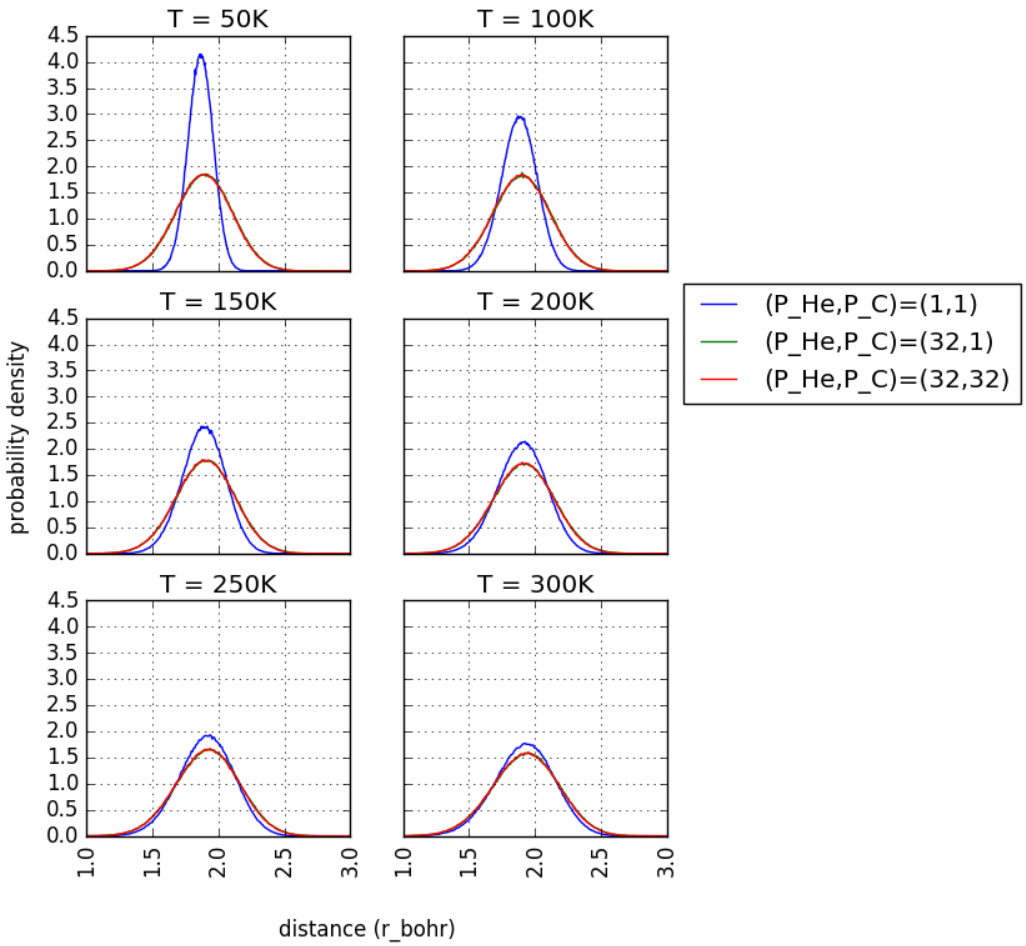

Figure 31: *Neutral dimer ($He_2@C_{60}$) helium atom position (he_dist) distribution according to the "FULL_NEW_NEUTRAL" model. The $(P_{He}, P_C) = (32, 1)$ and $(P_{He}, P_C) = (32, 32)$ curves are on top of each other because the helium atom distribution is not much affected by P_C.*

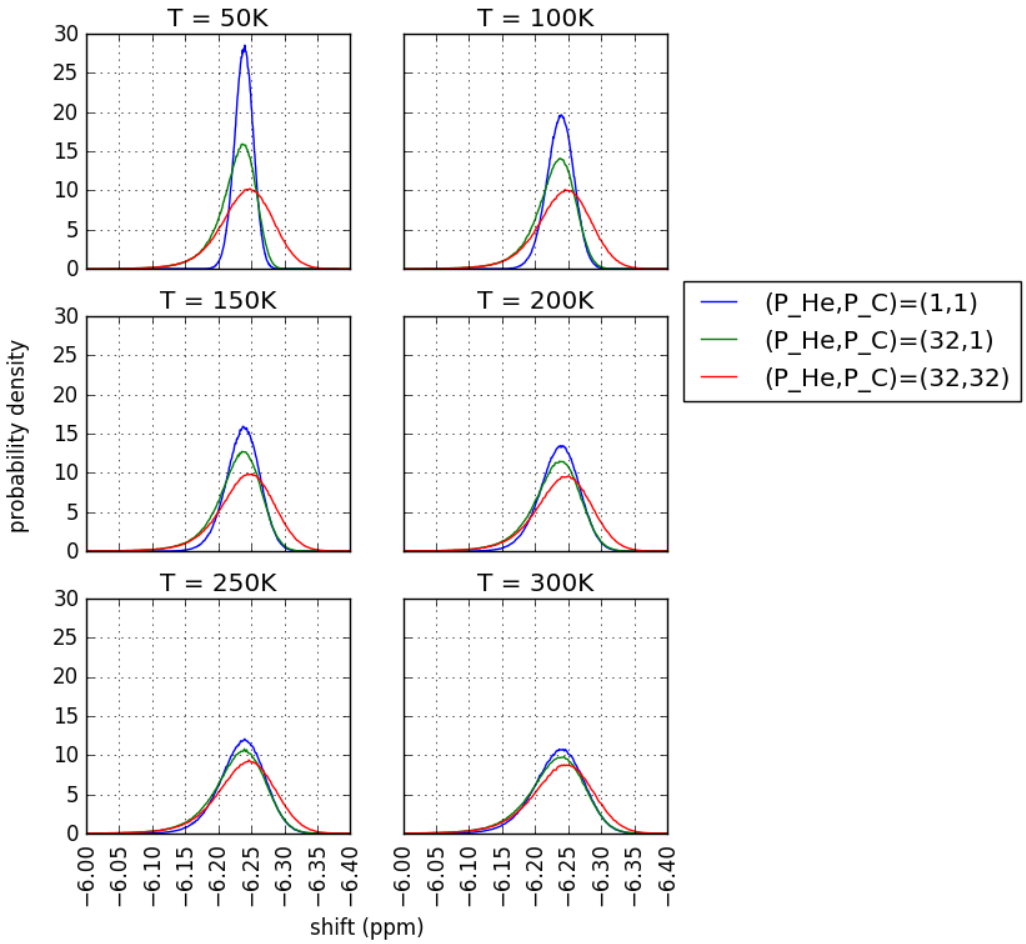

Figure 32: *Neutral dimer ($He_2@C_{60}$) total shift (total_shift) distribution according to the "FULL_NEW_NEUTRAL" model. Here both Trotter numbers are important.*

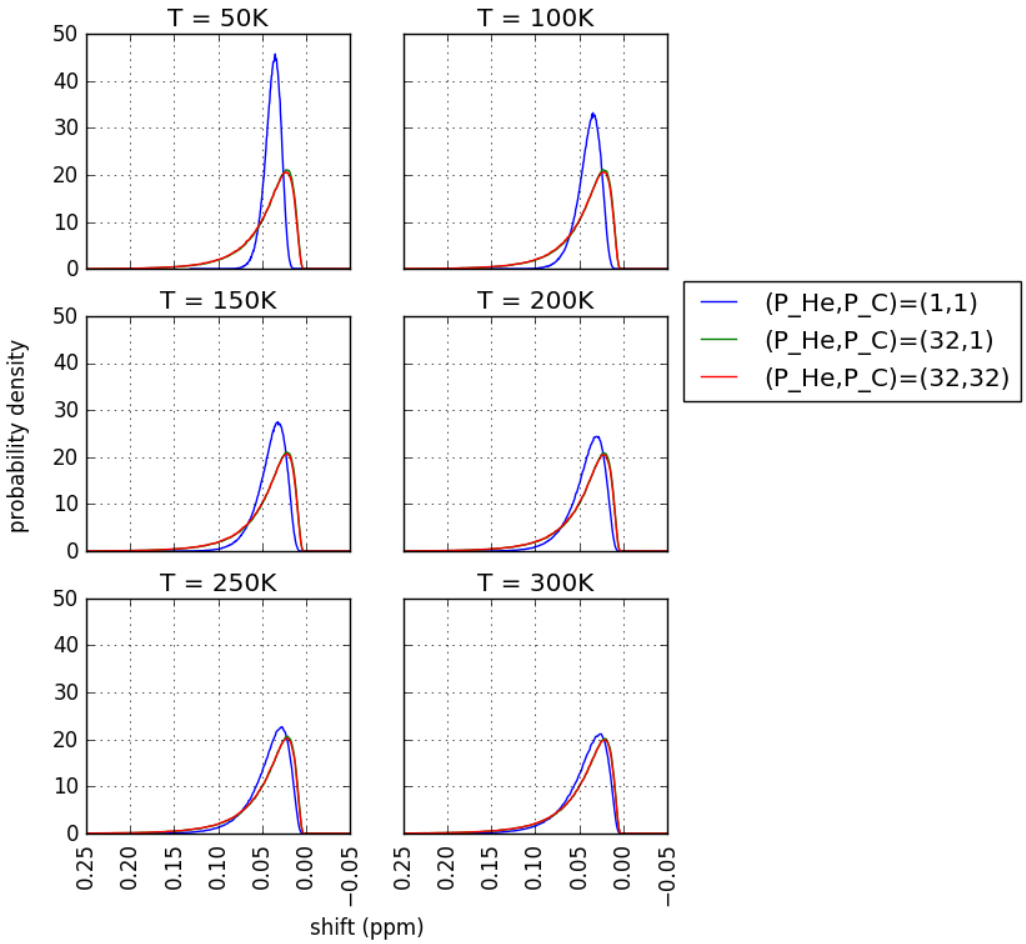

Figure 33: *Neutral dimer ($He_2@C_{60}$) helium-helium shift (he_he_shift) distribution according to the "FULL_NEW_NEUTRAL" model. The $(P_{He}, P_C) = (32, 1)$ and $(P_{He}, P_C) = (32, 32)$ curves are on top of each other, i.e. P_{He} is more important because it determines the helium atom distribution.*

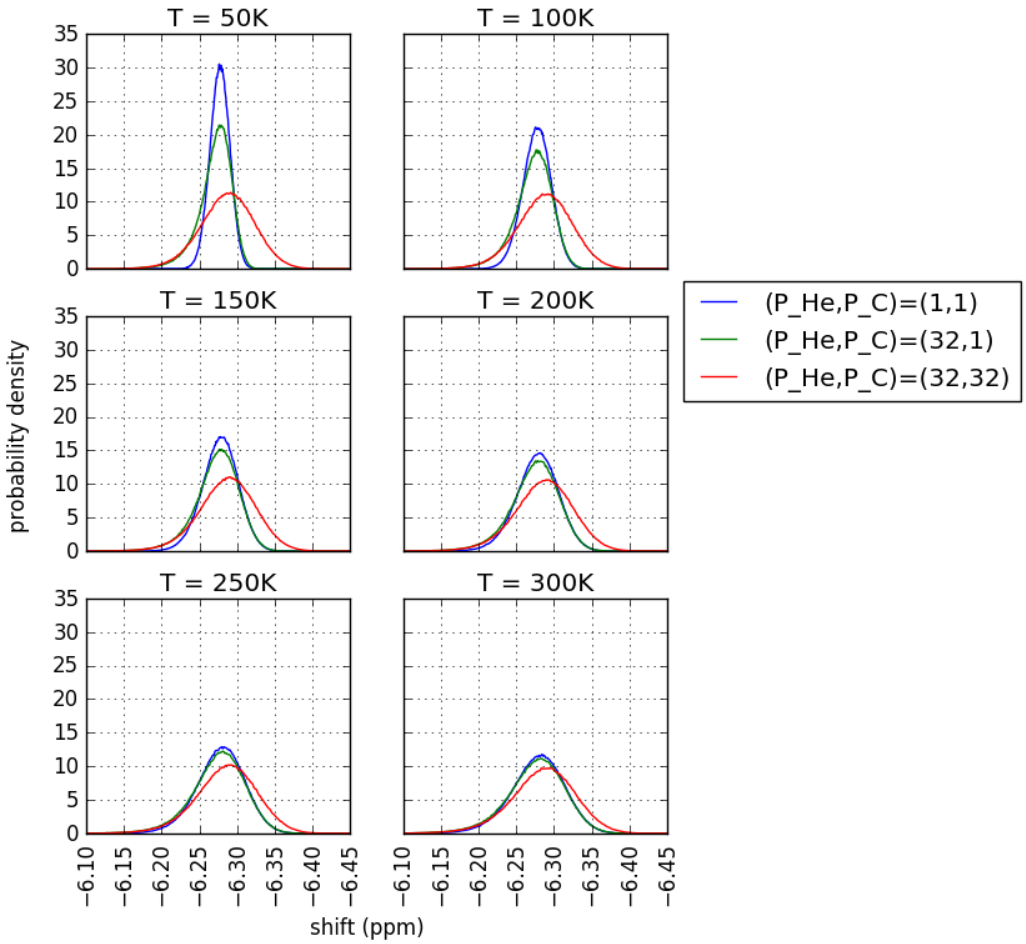

Figure 34: *Neutral dimer ($He_2@C_{60}$) helium-carbon shift (he_c_shift) distribution according to the "FULL_NEW_NEUTRAL" model. Here both Trotter numbers are important.*

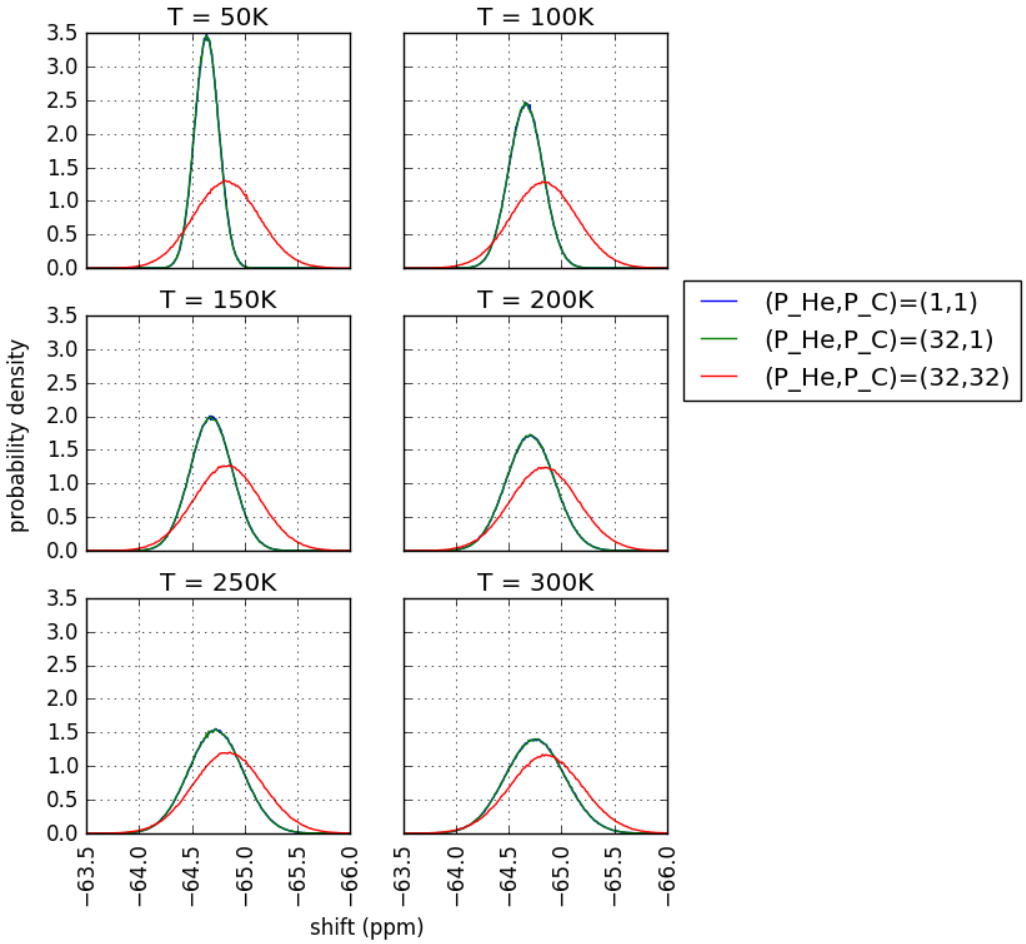

Figure 35: *Anion monomer* ($He_1@C_{60}^{6-}$) *total shift* (*total_shift*) *distribution according to the "FULL_NEW_ANION" model. The* $(P_{He}, P_C) = (1, 1)$ *and* $(P_{He}, P_C) = (32, 1)$ *curves are on top of each other.*

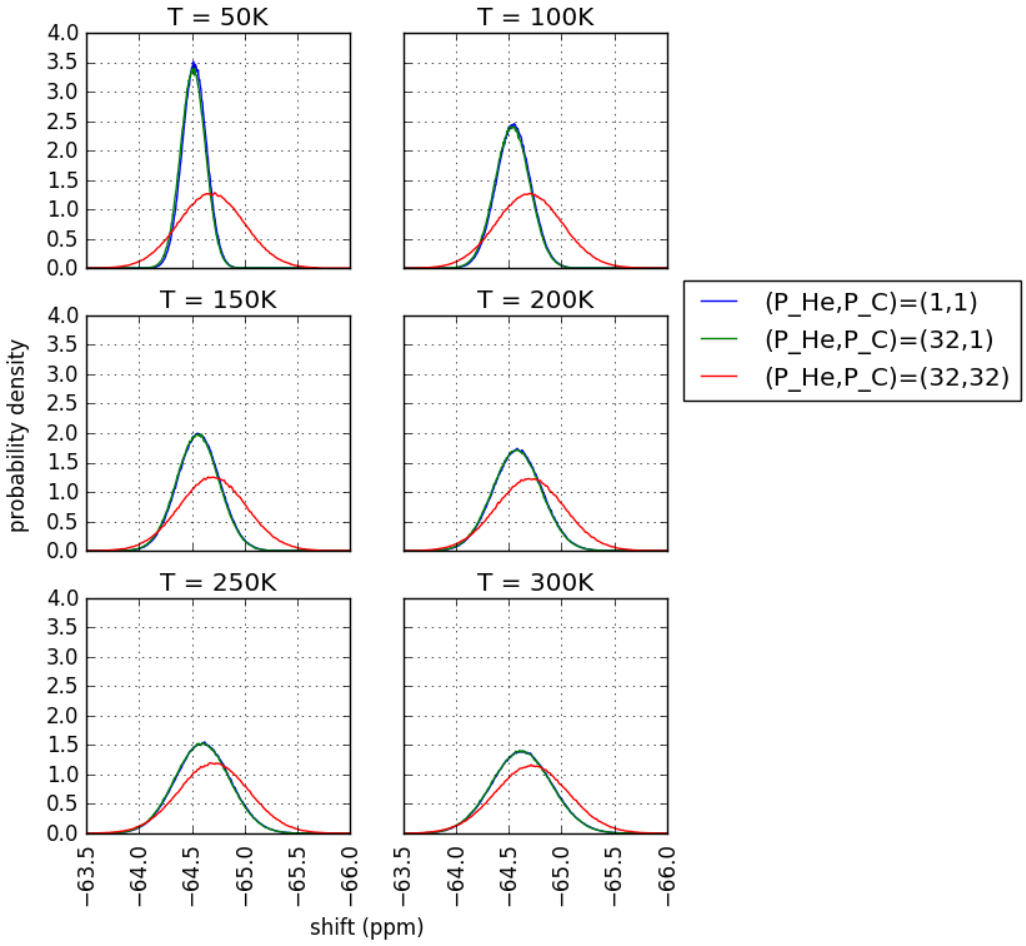

Figure 36: *Anion dimer ($He_2@C_{60}^{6-}$) total shift (total_shift) distribution according to the "FULL_NEW_ANION" model. The $(P_{He}, P_C) = (1, 1)$ and $(P_{He}, P_C) = (32, 1)$ curves are on top of each other. The helium-helium shift is not well visible here because it is so much weaker than the shielding from the fullerene, which is very strong for anion.*

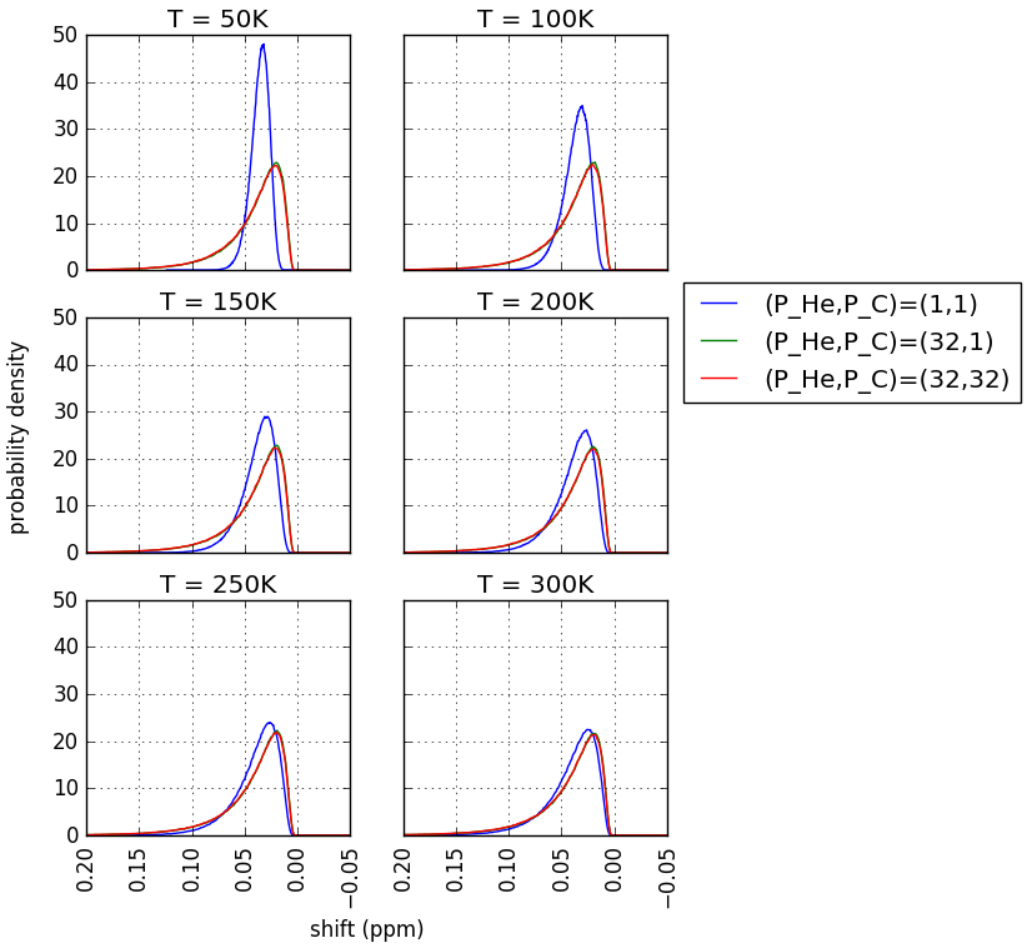

Figure 37: *Anion dimer ($He_2@C_{60}^{6-}$) helium-helium shift (he_he_shift) distribution according to the "FULL_NEW_ANION" model. The $(P_{He}, P_C) = (32, 1)$ and $(P_{He}, P_C) = (32, 32)$ curves are on top of each other.*

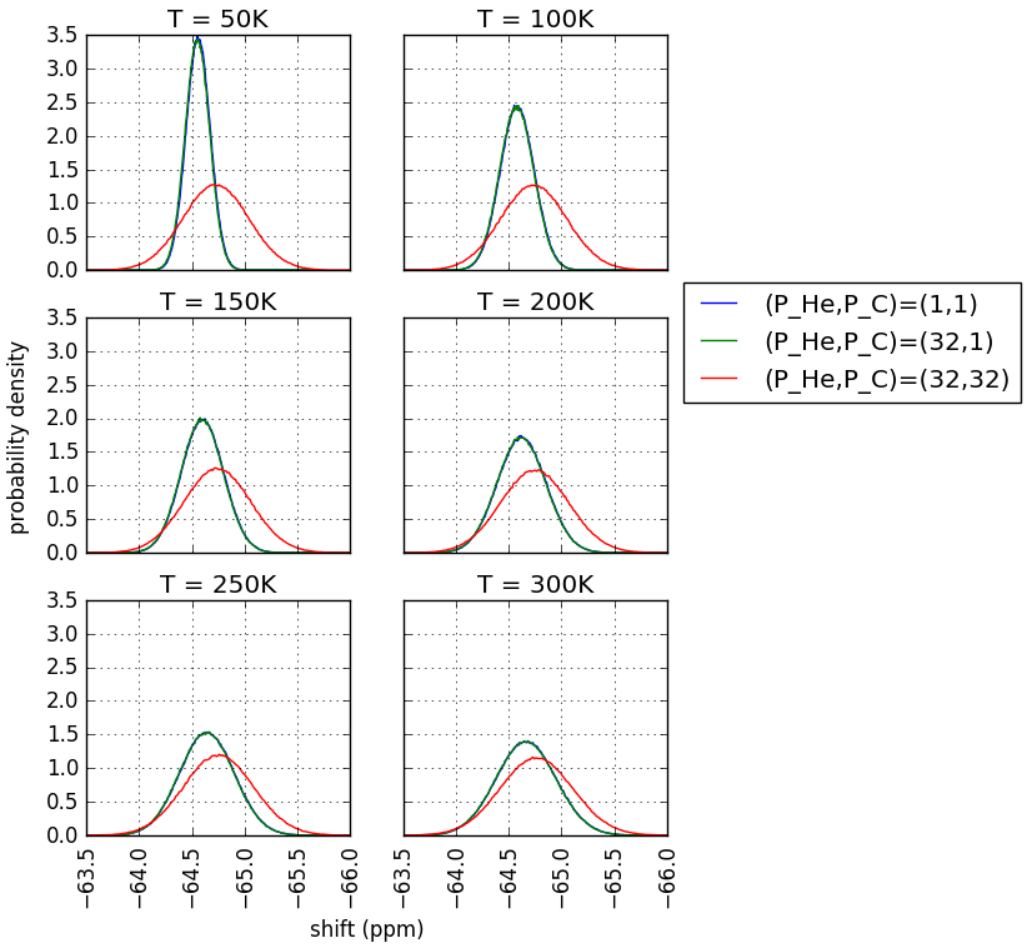

Figure 38: *Anion dimer ($He_2@C_{60}^{6-}$) helium-carbon shift (he_c_shift) distribution according to the "FULL_NEW_ANION" model. The $(P_{He}, P_C) = (1, 1)$ and $(P_{He}, P_C) = (32, 1)$ curves are on top of each other.*

B Source code

This chapter lists the source code for the three programs that were described in chapter 4:

1. The simulation program pimc.x, consisting of files: pimc.f90, simulation.f90, pimc_system.f90, randomgen.f90, pair_potential.f90 and constants.f90.

2. One version of postprocessing program postproc.x, consisting of files: postproc.cpp, distance.cpp and constants.cpp.

3. The time series analysis program ts_ana.x, consisting of files: ts_ana.cpp, stat_ineff.cp and autocorrelation.cpp. This program additionally requires the JsonCpp[37] and FFTW[38] libraries.

B.1 Simulation code

Contents of file pimc.f90:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This is a program for simulating Buckminsterfullerene with one or two helium
! atoms inside it. The simulation type is path integral Monte Carlo (PIMC).
!
! INPUT:
!     The program takes its input as command line arguments, where the arguments
!     are (in order):
!         1. Temperature in kelvins. Must be a floating point value e.g.
!            "300.0", where the ".0" is required. Using "300" will not work
!             correctly.
!         2. Trotter number for the helium atoms. Must be of form 2^n, e.g. "8"
!            or "32" are ok while "7" is not.
!         3. Trotter number for the carbon atoms. Must be of form 2^n.
!         4. Number of helium atoms. "1" or "2"
!         5. Number of datapoints. This is how many datapoints will be
!            collected. For example to collect a million datapoints use
!            "1000000".
!         6. Simulation type. An integer determining what kind of simulation
!            will be run. E.g. "0". See "SIMULATION TYPES" below for options.
!         7. Simulation time. This is approximately how many seconds the
!            simulation will take. For example if simulation time is "3600",
!            the simulation will run for one hour.
!         8. Output filename. The program will save the results in this file.
!            See "OUTPUT" below for details.
!     The program will not check all of its input so input that does not comply
!     with the above requirements may produce undefined behaviour. For example
!     the requirement that the trotter numbers be of form 2^n is not checked but
!     the program will assume this to be true. The program will print the input
!     values during startup so that they can be confirmed.
!
! OUTPUT:
!     The program will produce a binary file that consists of a 48 byte header
!     followed by the data. The structure is:
!         ------------------------------------------------------------
!         bytes | datatype        | contains
!         ------------------------------------------------------------
!         0-15  | character array | Version name.
!         16-23 | double          | temperature
!         24-27 | integer         | helium trotter number
!         28-31 | integer         | carbon trotter number
!         32-35 | integer         | number of helium atoms
!         36-39 | integer         | datapoints
!         40-43 | integer         | simulation type
!         44-47 | integer         | simulation time
!         48-   | double array    | raw simulation data
```

```
!     The "raw simulation data" contains the specified number of "datapoints".
!     Each datapoint consists of all the atom positions in the first
!     "time slice". The the atom positions in turn consist of the x, y and z
!     co-ordinates saved as doubles. In each datapoint the helium positions are
!     saved before the carbon positions. In simulation types 1, 2 and 3 the
!     carbon atom positions are not saved since they do not change.
!
! SIMULATION TYPES:
!     0 = "FULL": This will simulate the whole system (both helium atoms and
!         carbon atoms.)
!     1 = "RIGID": This is the same as "FULL" except the carbon atoms will not
!         move.
!     2 = "SPHERICAL": This simulates the helium atoms in an effective
!         spherically symmetrical potential. This replicates the system that
!         was studied in Katja Hyvönen's thesis (2015)[Hyvönen].
!     3 = "C70_RIGID": This is the same as "RIGID" except the fullerene
!         structure has been modified to approximately match C70 fullerene.
!         This structure has not been optimized in any way, and the pair
!         potential between Helium and Carbon may not have been optimized for
!         use case. This simulation type should probobaly not be used.
!     4 = "FULL_NEW_NEUTRAL": Same as "FULL", except the pair potential form
!         between helium and carbon atoms is changed.
!     5 = "FULL_NEW_ANION": Same as "FULL", exept the pair potential form
!         between helium and carbon atoms is changed. This force field attempts
!         to simulate anion (-6) fullerene.
!
! General structure of program:
!     The program can be divided to two phases: initialization and data
!     collection.
!         In the initialization phase the program reads the input and
!     initializes the requested simulation. This includes initializing the atom
!     positions, determining the best trial move type for each atom group,
!     optimizing the trial moves, calculatating the cpu-time required by the
!     selected trial moves, and determining how many trial moves of each type
!     should be made between each datapoint. Also the output file is opened and
!     the header is written to it.
!         In the data collection phase the program runs the optimized
!     simulation and writes the datapoints to the output file.
!
! Modules:
!     This program consists of six modules: "pimc.f90" (this file),
!     "simulation.f90", "pimc_system.f90", "constants.f90", "pair_potential.f90"
!     and "randomgen.f90".
!         The randomgen module contains different procedures for random number
!     generation and random choices.
!         The pimc_system contains set of tools which facilitate setting up an
!     PIMC simulation. This module is oblivious to what kind of molecular
!     system is studied, and instead offers a framework that needs to be
!     complemented with the molecular information, which includes the atoms
!     arranged to "atom groups" and the interactions that are experienced by
!     those groups. The pimc_system module takes care of most of the
!     non-molecule specific work. It decides what kind of trial moves are called
!     to get the best result in the given cpu time per datapoint.
!         The pimc_system uses three trial move types single-slice, bisection,
!     and modified bisection method. For each atom group one trial move type is
!     selected, and this trial move is then optimized to get fast convergence.
!     The cpu-time of each trial move is then estimated, and during the
!     simulation cpu-time will be evenly divided between the atom groups.
!          The "simulation.f90" module contains procedures to set up different
!     simulations. It uses the framework of "pimc_system.f90" to set up and
!     run the simulations. It also uses information from "constants.f90" and
!     "pair_potential.f90", which contain some molecule specific constants and
!     potential functions.
!          The "pimc.f90" file reads input from the user, initializes one of the
!     simulations defined in "simulatios.f90" and runs it, saving the results to
!     an output file specified by the user.
!
!
! About force fields and fullerene structure:
!     The He-Cforce fields and fullerene structure were provided by (,,,?)
!
!     The helium-helim force field and spherically symmetric force field used
!     used in the SPHERICAL simulation type are from [Hyvönen].
!
!     This program uses the "Extended Wu force field" [Ceulemans] to simulate
!     the C_60 fullerene.
!
!
! About the trial moves:
!     The program uses single slice moves and bisection moves, that are
!     described in many sources [Leino][Militzer]. The program handles the
!     situations where different number of "time slices" are used for different
!     atom groups with the method presented by Yimin Li & William H. Miller
```

```
!     [Li & Miller].
!
!
! Other info:
!     This program was written by Markku Alamäki in 2017-2018 as a part of his
!     thesis at the University of Oulu. This program tries to comply with the
!     Fortran 2003 standard.
!
!
! References:
!     [Ceulemans] Ceulemans, Arnout & C. Titeca, Bruno & F. Chibotaru, Liviu &
!         Vos, Ingrid & Fowler, Patrick. (2001). Complete Bond Force Fields for
!         Trivalent and Deltahedral Cages: Group Theory and Applications to
!         Cubane, Closo-dodecaborane, and Buckminsterfullerene. J. Phys. Chem.
!         A. 105. 8284-8295. 10.1021/jp0036792.
!     [Leino] M. Leino; Finite-Temperature Quantum Statistics of a Few Confined
!         Electrons and Atoms - Path-Integral Approach; (2007)
!     [Militzer] B. Militzer; Path integral Monte Carlo simulations of hot dense
!         hydrogen; (2000)
!     [Hyvönen] K. Hyvönen; Heliumatomin ja -dimeerin NMR kemialliset siirtymät
!         C_{60}-fullereenin muodostamassa keskeispotentiaalissa
!         polkuintegraali-Monte Carlo menetelmällä (2015)
!     [Li & Miller] Y. Li and W. H. Miller; Different time slices for different
!         degrees of freedom in Feynman path integration.
!     [He-C-pair potentials] (,,,where are these from/who made them/how)
!     [FullereneStructure] (,,,and this?)


program pimc
    use :: simulation
    implicit none

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Variables
!

    integer, parameter :: FULL_SIM = 0
    integer, parameter :: HE_ONLY_SIM = 1
    integer, parameter :: CAVITY_SIM = 2
    integer, parameter :: C70_RIGID_SIM = 3
    integer, parameter :: FULL_SIM_NEW_NEUTRAL = 4
    integer, parameter :: FULL_SIM_NEW_ANION = 5
    integer, parameter :: POSITION_OUT = 25
    integer, parameter :: STR_LEN = 256

    ! Input help variables
    integer :: arg_length
    character(len=1024) :: input_buffer

    ! Input values
    double precision :: temperature
    integer :: he_trotter_n, c_trotter_n, he_atoms
    integer :: datapoints
    integer :: simulation_type
    double precision :: time_per_datapoint
    integer :: total_simulation_time
    character(len=STR_LEN) :: output_filename

    ! Loop variables
    integer :: i, j, l

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Read input
!

    arg_length = command_argument_count()
    if (arg_length .EQ. 8) then
        call get_command_argument(1, input_buffer)
        read(input_buffer, fmt='␣(F20.10)') temperature
        call get_command_argument(2, input_buffer)
        read(input_buffer, fmt='(i20)') he_trotter_n
        call get_command_argument(3, input_buffer)
        read(input_buffer, fmt='(i20)') c_trotter_n
        call get_command_argument(4, input_buffer)
        read(input_buffer, fmt="(i20)") he_atoms
        call get_command_argument(5, input_buffer)
        read(input_buffer, fmt="(i20)") datapoints
        call get_command_argument(6, input_buffer)
        read(input_buffer, fmt="(i20)") simulation_type
        call get_command_argument(7, input_buffer)
        read(input_buffer, fmt="(i20)") total_simulation_time
        call get_command_argument(8, output_filename)
```

```
        print *, "INPUT␣VALUES:␣"
        print *, "␣␣␣␣temperature:␣␣␣␣␣␣", temperature
        print *, "␣␣␣␣he_trotter_n:␣␣␣␣␣", he_trotter_n
        print *, "␣␣␣␣c_trotter_n:␣␣␣␣␣␣", c_trotter_n
        print *, "␣␣␣␣he_atoms:␣␣␣␣␣␣␣␣", he_atoms
        print *, "␣␣␣␣datapoints:␣␣␣␣␣␣␣", datapoints
        print *, "␣␣␣␣simulation_type:␣␣", simulation_type
        print *, "␣␣␣␣simulation_time:␣␣", total_simulation_time
        print *, "␣␣␣␣output_file:␣␣␣␣␣␣", output_filename
    else
        print *, "Input␣not␣understood,␣exiting..."
        error stop
    end if


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Initialize and run simulation
!
    print *, "Initializing"
    time_per_datapoint = dble(total_simulation_time) / datapoints
    if (simulation_type == FULL_SIM) then
        selected_he_c_pair_potential => he_c_potential
        call initialize(temperature, he_atoms, he_trotter_n, c_trotter_n, time_per_datapoint)
    else if (simulation_type == HE_ONLY_SIM) then
        call he_only_init(temperature, he_atoms, he_trotter_n, time_per_datapoint)
    else if (simulation_type == CAVITY_SIM) then
        call cavity_init(temperature, he_atoms, he_trotter_n, time_per_datapoint)
    else if (simulation_type == C70_RIGID_SIM) then
        call c70_rigid_init(temperature, he_atoms, he_trotter_n, time_per_datapoint)
    else if (simulation_type == FULL_SIM_NEW_NEUTRAL) then
        selected_he_c_pair_potential => he_c_potential_new_neutral
        call initialize(temperature, he_atoms, he_trotter_n, c_trotter_n, time_per_datapoint)
    else if (simulation_type == FULL_SIM_NEW_ANION) then
        selected_he_c_pair_potential => he_c_potential_new_anion
        call initialize(temperature, he_atoms, he_trotter_n, c_trotter_n, time_per_datapoint)
    else
        print *, "Invalid␣simulation_type"
        error stop
    end if

    print *, "Collecting␣data"
    ! Prepare data collection
    open(unit=POSITION_OUT, file=output_filename, access="stream", status="replace", form="unformatted")
    write(unit=POSITION_OUT) "Version␣B18_v5␣␣" ! 16 characters
    write(unit=POSITION_OUT) temperature, he_trotter_n, c_trotter_n, he_atoms
    write(unit=POSITION_OUT) datapoints, simulation_type, total_simulation_time


    ! Running simulation and collecting data
    do i = 1, datapoints
        ! Run simulation
        if ( &
            (simulation_type == FULL_SIM) .OR. &
            (simulation_type == FULL_SIM_NEW_NEUTRAL) .OR. &
            (simulation_type == FULL_SIM_NEW_ANION) &
        ) then
            call run_simulation()
        else if (simulation_type == HE_ONLY_SIM) then
            call he_only_run_simulation()
        else if (simulation_type == CAVITY_SIM) then
            call cavity_run_simulation()
        else if (simulation_type == C70_RIGID_SIM) then
            call c70_rigid_run_simulation()
        end if

        ! Collect data
        do j = 1, he_atoms
            do l = 1, 3
                write(unit=POSITION_OUT) he_r_values(l, 1, j)
            end do
        end do
        if ( &
            (simulation_type == FULL_SIM) .OR. &
            (simulation_type == FULL_SIM_NEW_NEUTRAL) .OR. &
            (simulation_type == FULL_SIM_NEW_ANION) &
        ) then
            do j = 1, c_atoms
                do l = 1, 3
                    write(unit=POSITION_OUT) c_r_values(l, 1, j)
                end do
            end do
        end if
    end do
```

```
    print *, "Simulation finished:"
    call print_info()

    close(unit=POSITION_OUT)

    stop
end program pimc
```

Contents of file simulation.f90:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! SIMULATION MODULE
!     This module contains necessary methods to run simulations of three types:
!         - FULL: Simulate all atoms
!         - HE ONLY: Simulate only helium atoms; carbon atoms stay fixed
!         - CAVITY: Simulation in spherical field
!         -
!
!     Modules used by this module:
!         - pimc_system: The actual core around which all simulations are built.
!         - constants: Numerical constants, structure of fullerene etc..
!         - pair_potential: Potential energy functions.
!         - randomgen
!

module simulation
    use, intrinsic :: ISO_Fortran_env
    use :: pimc_system
    use :: pair_potential
    use :: randomgen
    use :: constants
    implicit none

    ! Full simulation methods
    public :: initialize, run_simulation
    private :: he_he_eval, he_c_eval, fullerene_eval, fullerene_pot
    private :: get_center_position, move_to_center

    ! He only simulation methods
    public :: he_only_init, he_only_run_simulation
    private :: he_only_eval

    ! Cavity simulation methods
    public :: cavity_init, cavity_run_simulation
    private :: cavity_eval

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! GLOBAL VARIABLES AND CONSTANTS
!     - these may or may not be used depending on the selected simulation type
!

    ! Constants
    integer, private :: he_trotter_n, c_trotter_n
    integer, private :: he_atoms
    integer, parameter, public :: C_ATOMS = 60
    integer, parameter, public :: HELIUM_INDEX = 1, CARBON_INDEX = 2
    integer, parameter, private :: INTERACTION_FULLERENE = 1, &
        INTERACTION_HE_HE = 2, INTERACTION_HE_C = 3

    ! Variables
    double precision, public, dimension (:,:,:), allocatable, target :: &
        he_r_values, c_r_values
    procedure (he_c_potential), pointer :: selected_he_c_pair_potential

contains


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FULL SIMULATION; initialize and run functions
!

    ! Initialize full simulation
    subroutine initialize(temperature, he_atoms_, he_trotter_n_, c_trotter_n_, time_per_datapoint)
        integer, parameter :: NUMBER_OF_INTERACTIONS = 3
        integer, parameter :: NUMBER_OF_GROUPS = 2
        double precision :: temperature, beta
        double precision, dimension(3) :: r
        integer :: he_atoms_, he_trotter_n_, c_trotter_n_, i, j
        double precision :: time_per_datapoint
        type (AtomGroup), dimension(NUMBER_OF_GROUPS) :: groups
        type (interaction), dimension(NUMBER_OF_INTERACTIONS) :: interactions
```

```fortran

        ! Temporary value
        beta = 1 / (KELVIN_TO_HARTREE * temperature)

        ! Set constants
        he_atoms = he_atoms_
        he_trotter_n = he_trotter_n_
        c_trotter_n = c_trotter_n_

        ! Initialize Helium positions
        allocate (he_r_values(3, he_trotter_n, he_atoms))
        do i = 1, he_atoms
            if (i == 1) then
                r = (/1.9, 0.0, 0.0/)
            else
                r = (/-1.9, 0.0, 0.0/)
            end if
            do j = 1, he_trotter_n
                he_r_values(:, j, i) = r
            end do
        end do

        ! Initialize Carbon positions
        allocate (c_r_values(3, c_trotter_n, C_ATOMS))
        do i = 1, C_ATOMS
            do j = 1, c_trotter_n
                c_r_values(:, j, i) = FULLERENE_R_VALUES(:, i)
            end do
        end do

        ! Define interactions
        interactions(INTERACTION_FULLERENE)%eval => fullerene_eval
        interactions(INTERACTION_HE_C)%eval => he_c_eval
        interactions(INTERACTION_HE_HE)%eval => he_he_eval

        ! HELIUM GROUP
        groups(HELIUM_INDEX)%name = "Helium atoms."
        groups(HELIUM_INDEX)%trotter = he_trotter_n
        groups(HELIUM_INDEX)%nat = he_atoms_
        groups(HELIUM_INDEX)%r_values => he_r_values

        ! There are two interactions acting on helium group
        groups(HELIUM_INDEX)%nint = 2
        allocate(groups(HELIUM_INDEX)%interactions(2))
        groups(HELIUM_INDEX)%interactions(1) = INTERACTION_HE_HE
        groups(HELIUM_INDEX)%interactions(2) = INTERACTION_HE_C

        ! Other values
        groups(HELIUM_INDEX)%dx_max = 0.01
        groups(HELIUM_INDEX)%mass = MASS_HE3_ATOMIC

        ! CARBON GROUP
        groups(CARBON_INDEX)%name = "C-60 fullerene."
        groups(CARBON_INDEX)%trotter = c_trotter_n
        groups(CARBON_INDEX)%nat = C_ATOMS
        groups(CARBON_INDEX)%r_values => c_r_values

        ! There are two interactions acting on carbon group
        groups(CARBON_INDEX)%nint = 2
        allocate(groups(CARBON_INDEX)%interactions(2))
        groups(CARBON_INDEX)%interactions(1) = INTERACTION_FULLERENE
        groups(CARBON_INDEX)%interactions(2) = INTERACTION_HE_C

        ! Other values
        groups(CARBON_INDEX)%dx_max = 0.01
        groups(CARBON_INDEX)%mass = MASS_C_ATOMIC

        call pimc_system_init(beta, NUMBER_OF_GROUPS, groups, NUMBER_OF_INTERACTIONS, interactions, time_per_datapoint)

    end subroutine initialize

    ! Run full simulation for some number of rounds
    subroutine run_simulation()
        call relax_system()
        call move_to_center()
    end subroutine run_simulation


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FULL SIMULATION; Evaluation of interactions
!

    function he_he_eval(group_index, atom_index, slice_index, r_new)
```

```
        double precision :: he_he_eval, ed
        integer, intent (in) :: group_index, atom_index, slice_index
        double precision, dimension(3), intent (in) :: r_new
        double precision, dimension(3) :: r_cur, r_other


        r_cur = get_r(group_index, atom_index, slice_index)

        if (group_index .NE. HELIUM_INDEX) then
            print *, "he_he_eval called with wrong group_index."
            error stop
        end if

        ed = 0d0
        if (he_atoms == 2) then
            if (atom_index == 1) then
                r_other = get_r(HELIUM_INDEX, 2, slice_index)
            else
                r_other = get_r(HELIUM_INDEX, 1, slice_index)
            end if
            ed = ed + ( &
                he_he_potential(sqrt(sum((r_new - r_other) ** 2)))&
                - he_he_potential(sqrt(sum((r_cur - r_other) ** 2)))&
            )
        end if
        !print *, "he_he_eval", ed
        he_he_eval = ed
        return
    end function he_he_eval

    function he_c_eval(group_index, atom_index, slice_index, r_new)
        double precision :: he_c_eval, ed
        integer, intent (in) :: group_index, atom_index, slice_index
        double precision, dimension(3), intent (in) :: r_new
        double precision, dimension(3) :: r_cur, temp_vec
        integer :: i

        r_cur = get_r(group_index, atom_index, slice_index)

        ed = 0d0
        if (group_index == CARBON_INDEX) then
            do i = 1, he_atoms
                temp_vec = get_r(HELIUM_INDEX, i, slice_index)
                ed = ed + (&
                      selected_he_c_pair_potential(sqrt(sum((r_new - temp_vec) ** 2)))&
                    - selected_he_c_pair_potential(sqrt(sum((r_cur - temp_vec) ** 2)))&
                )
            end do
        else if (group_index == HELIUM_INDEX) then
            do i = 1, C_ATOMS
                temp_vec = get_r(CARBON_INDEX, i, slice_index)
                ed = ed + (&
                      selected_he_c_pair_potential(sqrt(sum((r_new - temp_vec) ** 2)))&
                    - selected_he_c_pair_potential(sqrt(sum((r_cur - temp_vec) ** 2)))&
                )
            end do
        end if
        he_c_eval = ed
        return
    end function he_c_eval

    function fullerene_eval(group_index, atom_index, slice_index, r_new)
        double precision :: fullerene_eval
        integer, intent (in) :: group_index, atom_index, slice_index
        double precision, dimension(3), intent (in) :: r_new
        double precision, dimension(3) :: r_cur

        if (group_index .NE. CARBON_INDEX) then
            print *, "fullerene_eval called with wrong group_index."
            error stop
        end if

        r_cur = get_r(CARBON_INDEX, atom_index, slice_index)

        fullerene_eval = fullerene_pot(atom_index, slice_index, r_new)&
                        -fullerene_pot(atom_index, slice_index, r_cur)
        !print *, "fullerene_eval", fullerene_eval
        return
    end function fullerene_eval


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FULL SIMULATION; helping function for evaluating potential energy
```

```
!

    ! Calculates energy (not the energy difference!) coming from those
    ! interactions that affect the carbon bead corresponding to a given
    ! atom_index and slice_index when that bead is moved to r0
    function fullerene_pot(atom_index, slice_index, r0)
        double precision :: fullerene_pot
        integer :: atom_index, slice_index

        double precision, dimension(3) :: r0, r1, r2, r3, r4, r5, r6, r7, r8, r9
        double precision, dimension(3) :: r_01, r_02, r_03, r_14, r_15, r_26, r_27, r_38, r_39
        double precision :: r01, r02, r03, r14, r15, r26, r27, r38, r39
        double precision :: dr01, dr02, dr03, dr14, dr15, dr26, dr27, dr38, dr39
        double precision :: a102, a103, a203, a014, a015, a026, a027, a038, a039, a415, a627, a839
        double precision :: da102, da103, da203, da014, da015, da026, da027, da038, da039, da415, da627, da839
        double precision :: energy

        ! Get r1,...
        r1 = get_r(CARBON_INDEX, C_NEIGHBOURS(1, atom_index), slice_index)
        r2 = get_r(CARBON_INDEX, C_NEIGHBOURS(2, atom_index), slice_index)
        r3 = get_r(CARBON_INDEX, C_NEIGHBOURS(3, atom_index), slice_index)
        r4 = get_r(CARBON_INDEX, C_NEIGHBOURS(4, atom_index), slice_index)
        r5 = get_r(CARBON_INDEX, C_NEIGHBOURS(5, atom_index), slice_index)
        r6 = get_r(CARBON_INDEX, C_NEIGHBOURS(6, atom_index), slice_index)
        r7 = get_r(CARBON_INDEX, C_NEIGHBOURS(7, atom_index), slice_index)
        r8 = get_r(CARBON_INDEX, C_NEIGHBOURS(8, atom_index), slice_index)
        r9 = get_r(CARBON_INDEX, C_NEIGHBOURS(9, atom_index), slice_index)

        ! Calculate r_01,...
        r_01 = r0 - r1
        r_02 = r0 - r2
        r_03 = r0 - r3
        r_14 = r1 - r4
        r_15 = r1 - r5
        r_26 = r2 - r6
        r_27 = r2 - r7
        r_38 = r3 - r8
        r_39 = r3 - r9

        ! Calculate r01,...
        r01 = sqrt(sum(r_01 ** 2))
        r02 = sqrt(sum(r_02 ** 2))
        r03 = sqrt(sum(r_03 ** 2))
        r14 = sqrt(sum(r_14 ** 2))
        r15 = sqrt(sum(r_15 ** 2))
        r26 = sqrt(sum(r_26 ** 2))
        r27 = sqrt(sum(r_27 ** 2))
        r38 = sqrt(sum(r_38 ** 2))
        r39 = sqrt(sum(r_39 ** 2))

        ! Calculate a102,...
        a102 = acos(sum( r_01 * r_02) / (r01 * r02))
        a103 = acos(sum( r_01 * r_03) / (r01 * r03))
        a203 = acos(sum( r_02 * r_03) / (r02 * r03))
        a014 = acos(sum(-r_01 * r_14) / (r01 * r14))
        a015 = acos(sum(-r_01 * r_15) / (r01 * r15))
        a026 = acos(sum(-r_02 * r_26) / (r02 * r26))
        a027 = acos(sum(-r_02 * r_27) / (r02 * r27))
        a038 = acos(sum(-r_03 * r_38) / (r03 * r38))
        a039 = acos(sum(-r_03 * r_39) / (r03 * r39))
        a415 = acos(sum( r_14 * r_15) / (r14 * r15))
        a627 = acos(sum( r_26 * r_27) / (r26 * r27))
        a839 = acos(sum( r_38 * r_39) / (r38 * r39))

        ! Calculate dr01,...
        dr01 = r01 - wu_hh_r
        dr02 = r02 - wu_ph_r
        dr03 = r03 - wu_ph_r
        dr14 = r14 - wu_ph_r
        dr15 = r15 - wu_ph_r
        dr26 = r26 - wu_hh_r
        dr27 = r27 - wu_ph_r
        dr38 = r38 - wu_hh_r
        dr39 = r39 - wu_ph_r

        ! Calculate da102,...
        da102 = a102 - wu_h_a
        da103 = a103 - wu_h_a
        da203 = a203 - wu_p_a
        da014 = a014 - wu_h_a
        da015 = a015 - wu_h_a
        da026 = a026 - wu_h_a
        da027 = a027 - wu_p_a
```

```fortran
        da038 = a038 - wu_h_a
        da039 = a039 - wu_p_a
        da415 = a415 - wu_p_a
        da627 = a627 - wu_h_a
        da839 = a839 - wu_h_a

        ! Calculate energy
        energy = 0

        ! Calculate energy, c1 and c2
        energy = energy + wu_c1 * dr01 ** 2
        energy = energy + wu_c2 * dr02 ** 2
        energy = energy + wu_c2 * dr03 ** 2
        energy = energy + wu_c2 * dr14 ** 2
        energy = energy + wu_c2 * dr15 ** 2
        energy = energy + wu_c1 * dr26 ** 2
        energy = energy + wu_c2 * dr27 ** 2
        energy = energy + wu_c1 * dr38 ** 2
        energy = energy + wu_c2 * dr39 ** 2

        ! Calculate energy, c3 and c4
        energy = energy + wu_c3 * r01 * r02 * da102 ** 2
        energy = energy + wu_c3 * r01 * r03 * da103 ** 2
        energy = energy + wu_c4 * r02 * r03 * da203 ** 2
        energy = energy + wu_c3 * r01 * r14 * da014 ** 2
        energy = energy + wu_c3 * r01 * r15 * da015 ** 2
        energy = energy + wu_c3 * r02 * r26 * da026 ** 2
        energy = energy + wu_c4 * r02 * r27 * da027 ** 2
        energy = energy + wu_c3 * r03 * r38 * da038 ** 2
        energy = energy + wu_c4 * r03 * r39 * da039 ** 2

        ! Calculate energy, c5 and c6
        energy = energy + 2 * wu_c5 * dr01 * dr02
        energy = energy + 2 * wu_c5 * dr01 * dr03
        energy = energy + 2 * wu_c6 * dr02 * dr03
        energy = energy + 2 * wu_c5 * dr01 * dr14
        energy = energy + 2 * wu_c5 * dr01 * dr15
        energy = energy + 2 * wu_c5 * dr02 * dr26
        energy = energy + 2 * wu_c6 * dr02 * dr27
        energy = energy + 2 * wu_c5 * dr03 * dr38
        energy = energy + 2 * wu_c6 * dr03 * dr39

        ! Calculate energy, c7 and c8
        energy = energy + 2 * wu_c7 * da103 * da203 * r03 * sqrt(r01 * r02)!102
        energy = energy + 2 * wu_c7 * da203 * da102 * r02 * sqrt(r01 * r03)!103
        energy = energy + 2 * wu_c8 * da102 * da103 * r01 * sqrt(r02 * r03)!203

        energy = energy + 2 * wu_c7 * da015 * da415 * r15 * sqrt(r01 * r14)!014
        energy = energy + 2 * wu_c7 * da014 * da415 * r14 * sqrt(r01 * r15)!015
        energy = energy + 2 * wu_c8 * da014 * da015 * r01 * sqrt(r14 * r15)!415

        energy = energy + 2 * wu_c7 * da027 * da627 * r27 * sqrt(r02 * r26)!026
        energy = energy + 2 * wu_c8 * da026 * da627 * r26 * sqrt(r02 * r27)!027
        energy = energy + 2 * wu_c7 * da026 * da027 * r02 * sqrt(r26 * r27)!627

        energy = energy + 2 * wu_c7 * da039 * da839 * r39 * sqrt(r03 * r38)!038
        energy = energy + 2 * wu_c8 * da038 * da839 * r38 * sqrt(r03 * r39)!039
        energy = energy + 2 * wu_c7 * da038 * da039 * r03 * sqrt(r38 * r39)!839

        ! Calculate energy, c9 and c10
        energy = energy + 2 * wu_c10 * (sqrt(sum((r0 - r4) ** 2)) - wu_h_r) ** 2!04
        energy = energy + 2 * wu_c10 * (sqrt(sum((r0 - r5) ** 2)) - wu_h_r) ** 2!05
        energy = energy + 2 * wu_c10 * (sqrt(sum((r0 - r6) ** 2)) - wu_h_r) ** 2!06
        energy = energy + 2 * wu_c9  * (sqrt(sum((r0 - r7) ** 2)) - wu_p_r) ** 2!07
        energy = energy + 2 * wu_c10 * (sqrt(sum((r0 - r8) ** 2)) - wu_h_r) ** 2!08
        energy = energy + 2 * wu_c9  * (sqrt(sum((r0 - r9) ** 2)) - wu_p_r) ** 2!09

        fullerene_pot = energy / 2
        return

    end function fullerene_pot


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FULL SIMULATION; centering the system
!

    ! Get average position of the carbon atoms
    function get_center_position()
        double precision, dimension(3)  :: get_center_position
        double precision, dimension(3) :: center_position
        integer :: i, j
        center_position = (/0.0, 0.0, 0.0/)
```

```
        do i = 1, C_ATOMS
            do j = 1, c_trotter_n
                center_position = center_position + c_r_values(:, j, i)
            end do
        end do
        get_center_position = center_position / (C_ATOMS * c_trotter_n)
    end function get_center_position

    ! Move the system to origo
    subroutine move_to_center()
        double precision, dimension(3) :: center_position
        integer :: i, j
        center_position = get_center_position()

        ! Move carbon atoms
        do i = 1, C_ATOMS
            do j = 1, c_trotter_n
                c_r_values(:, j, i) = c_r_values(:, j, i) - center_position
            end do
        end do

        ! Move helium atoms
        do i = 1, he_atoms
            do j = 1, he_trotter_n
                he_r_values(:, j, i) = he_r_values(:, j, i) - center_position
            end do
        end do

    end subroutine move_to_center

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! HE ONLY SIMULATION; initialize and run functions
!

    ! Initialize simulation where only he trial moves are used
    subroutine he_only_init(temperature, he_atoms_, he_trotter_n_, time_per_datapoint)
        integer, parameter :: NUMBER_OF_INTERACTIONS = 1
        integer, parameter :: NUMBER_OF_GROUPS = 1
        double precision :: temperature, beta, r
        integer :: he_atoms_, he_trotter_n_, i, j, k
        double precision :: time_per_datapoint
        type (AtomGroup), dimension(NUMBER_OF_GROUPS) :: groups
        type (interaction), dimension(NUMBER_OF_INTERACTIONS) :: interactions

        ! Temporary value
        !beta = HARTREE_TO_JOULE / (BOLTZMANN_CONSTANT * temperature)
        beta = 1 / (KELVIN_TO_HARTREE * temperature)

        ! Set constants
        he_atoms = he_atoms_
        he_trotter_n = he_trotter_n_
        c_trotter_n = 1

        ! Initialize Helium positions
        allocate (he_r_values(3, he_trotter_n, he_atoms))
        do i = 1, he_atoms
            if (i == 1) then
                r = 0.19
            else
                r = -0.19
            end if
            do j = 1, he_trotter_n
                do k = 1, 3
                    he_r_values(k, j, i) = r + 0.2 * random_float(-1.0d0, 1.0d0)
                end do
            end do
        end do

        ! Define interactions
        interactions(1)%eval => he_only_eval

        ! HELIUM GROUP
        groups(1)%name = "Helium atoms."
        groups(1)%trotter = he_trotter_n
        groups(1)%nat = he_atoms_
        groups(1)%r_values => he_r_values

        !
        groups(1)%nint = 1
        allocate(groups(1)%interactions(1))
        groups(1)%interactions(1) = 1

        ! Other values
```

```
        groups(1)%dx_max = 0.01
        !groups(1)%bead_constant = he_trotter_n * MASS_HE3_ATOMIC / (2 * beta ** 2)
        groups(1)%mass = MASS_HE3_ATOMIC

        call pimc_system_init(beta, NUMBER_OF_GROUPS, groups, NUMBER_OF_INTERACTIONS, interactions, time_per_datapoint)

    end subroutine he_only_init

    ! Run the simulation for a number of rounds
    subroutine he_only_run_simulation()
        call relax_system()
    end subroutine he_only_run_simulation


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! HE ONLY SIMULATION; ; Evaluation of interactions
!


    function he_only_eval(group_index, atom_index, slice_index, r_new)
        double precision :: he_only_eval, ed
        integer, intent (in) :: group_index, atom_index, slice_index
        double precision, dimension(3), intent (in) :: r_new
        double precision, dimension(3) :: r_cur, r_other
        integer :: i

        r_cur = get_r(group_index, atom_index, slice_index)

        ed = 0d0
        if (he_atoms == 2) then
            if (atom_index == 1) then
                r_other = get_r(HELIUM_INDEX, 2, slice_index)
            else
                r_other = get_r(HELIUM_INDEX, 1, slice_index)
            end if
            ed = ed + ( &
                he_he_potential(sqrt(sum((r_new - r_other) ** 2)))&
                - he_he_potential(sqrt(sum((r_cur - r_other) ** 2)))&
            )
        end if

        do i = 1, C_ATOMS
            ed = ed + (&
                  he_c_potential(sqrt(sum((r_new - FULLERENE_R_VALUES(:, i)) ** 2)))&
                - he_c_potential(sqrt(sum((r_cur - FULLERENE_R_VALUES(:, i)) ** 2)))&
            )
        end do

        he_only_eval = ed
        return
    end function he_only_eval



!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! CAVITY MODEL SIMULATION; initialize and run functions
!

    subroutine cavity_init(temperature, he_atoms_, he_trotter_n_, time_per_datapoint)
        integer, parameter :: NUMBER_OF_INTERACTIONS = 1
        integer, parameter :: NUMBER_OF_GROUPS = 1
        double precision :: temperature, beta, r
        integer :: he_atoms_, he_trotter_n_, i, j, k
        double precision :: time_per_datapoint
        type (AtomGroup), dimension(NUMBER_OF_GROUPS) :: groups
        type (interaction), dimension(NUMBER_OF_INTERACTIONS) :: interactions

        ! Temporary value
        !beta = HARTREE_TO_JOULE / (BOLTZMANN_CONSTANT * temperature)
        beta = 1 / (KELVIN_TO_HARTREE * temperature)

        ! Set constants
        he_atoms = he_atoms_
        he_trotter_n = he_trotter_n_
        !c_trotter_n = 1

        ! Initialize Helium positions
        allocate (he_r_values(3, he_trotter_n, he_atoms))
        do i = 1, he_atoms
            if (i == 1) then
                r = 0.19
            else
                r = -0.19
```

```
            end if
            do j = 1, he_trotter_n
                do k = 1, 3
                    he_r_values(k, j, i) = r + 0.2 * random_float(-1.0d0, 1.0d0)
                end do
            end do
        end do

        ! Define interactions
        interactions(1)%eval => cavity_eval

        ! HELIUM GROUP
        groups(1)%name = "Helium␣atoms."
        groups(1)%trotter = he_trotter_n
        groups(1)%nat = he_atoms_
        groups(1)%r_values => he_r_values

        !
        groups(1)%nint = 1
        allocate(groups(1)%interactions(1))
        groups(1)%interactions(1) = 1

        ! Other values
        groups(1)%dx_max = 0.01
        !groups(1)%bead_constant = he_trotter_n * MASS_HE3_ATOMIC / (2 * beta ** 2)
        groups(1)%mass = MASS_HE3_ATOMIC

        call pimc_system_init(beta, NUMBER_OF_GROUPS, groups, NUMBER_OF_INTERACTIONS, interactions, time_per_datapoint)

    end subroutine cavity_init

    subroutine cavity_run_simulation()
        call relax_system()
    end subroutine cavity_run_simulation

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! CAVITY MODEL SIMULATION; ; Evaluation of interactions
!

    function cavity_eval(group_index, atom_index, slice_index, r_new)
        double precision :: cavity_eval, ed
        double precision, dimension(3), intent (in) :: r_new
        integer, intent (in) :: group_index, atom_index, slice_index
        double precision, dimension(3) :: r_cur, r_other

        r_cur = get_r(group_index, atom_index, slice_index)

        ed = 0d0
        if (he_atoms == 2) then
            if (atom_index == 1) then
                r_other = get_r(HELIUM_INDEX, 2, slice_index)
            else
                r_other = get_r(HELIUM_INDEX, 1, slice_index)
            end if
            ed = ed + ( &
                he_he_potential(sqrt(sum((r_new - r_other) ** 2)))&
                - he_he_potential(sqrt(sum((r_cur - r_other) ** 2)))&
            )
        end if

        ed = ed + fullerene_potential(r_new) - fullerene_potential(r_cur)

        cavity_eval = ed

        return
    end function cavity_eval


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! C70 RIGID SIMULATION; initialize and run functions
!

! Initialize C70 simulation
    subroutine c70_rigid_init(temperature, he_atoms_, he_trotter_n_, time_per_datapoint)
        integer, parameter :: NUMBER_OF_INTERACTIONS = 1
        integer, parameter :: NUMBER_OF_GROUPS = 1
        double precision :: temperature, beta, r
        integer :: he_atoms_, he_trotter_n_, i, j, k
        double precision :: time_per_datapoint
        type (AtomGroup), dimension(NUMBER_OF_GROUPS) :: groups
        type (interaction), dimension(NUMBER_OF_INTERACTIONS) :: interactions

        ! Temporary value
```

```
        beta = 1 / (KELVIN_TO_HARTREE * temperature)

        ! Set constants
        he_atoms = he_atoms_
        he_trotter_n = he_trotter_n_
        c_trotter_n = 1

        ! Initialize Helium positions
        allocate (he_r_values(3, he_trotter_n, he_atoms))
        do i = 1, he_atoms
            if (i == 1) then
                r = 0.19
            else
                r = -0.19
            end if
            do j = 1, he_trotter_n
                do k = 1, 3
                    he_r_values(k, j, i) = r + 0.2 * random_float(-1.0d0, 1.0d0)
                end do
            end do
        end do

        ! Define interactions
        interactions(1)%eval => c70_rigid_eval

        ! HELIUM GROUP
        groups(1)%name = "Helium atoms."
        groups(1)%trotter = he_trotter_n
        groups(1)%nat = he_atoms_
        groups(1)%r_values => he_r_values

        !
        groups(1)%nint = 1
        allocate(groups(1)%interactions(1))
        groups(1)%interactions(1) = 1

        ! Other values
        groups(1)%dx_max = 0.01
        groups(1)%mass = MASS_HE3_ATOMIC

        call pimc_system_init(beta, NUMBER_OF_GROUPS, groups, NUMBER_OF_INTERACTIONS, interactions, time_per_datapoint)

    end subroutine c70_rigid_init

    ! Run the simulation for a number of rounds
    subroutine c70_rigid_run_simulation()
        call relax_system()
    end subroutine c70_rigid_run_simulation


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! C70 RIGID SIMULATION; evaluation of interactions
!

    function c70_rigid_eval(group_index, atom_index, slice_index, r_new)
        double precision :: c70_rigid_eval, ed
        integer, intent (in) :: group_index, atom_index, slice_index
        double precision, dimension(3), intent (in) :: r_new
        double precision, dimension(3) :: r_cur, r_other
        integer :: i

        r_cur = get_r(group_index, atom_index, slice_index)

        ed = 0d0
        if (he_atoms == 2) then
            if (atom_index == 1) then
                r_other = get_r(HELIUM_INDEX, 2, slice_index)
            else
                r_other = get_r(HELIUM_INDEX, 1, slice_index)
            end if
            ed = ed + ( &
                he_he_potential(sqrt(sum((r_new - r_other) ** 2)))&
                - he_he_potential(sqrt(sum((r_cur - r_other) ** 2)))&
            )
        end if

        do i = 1, 70
            ed = ed + (&
                  he_c_potential(sqrt(sum((r_new - C70_R_VALUES(:, i)) ** 2)))&
                - he_c_potential(sqrt(sum((r_cur - C70_R_VALUES(:, i)) ** 2)))&
            )
```

```
        end do
737
        c70_rigid_eval = ed
        return
    end function c70_rigid_eval

742


end module simulation
```

Contents of file pimc_system.f90:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
2 ! PIMC_SYSTEM MODULE
!     This module facilitates PIMC simulations.
!
!     Features included:
!          - Different number of time slices for atoms.
7 !          - automatic optimization of dx_maxes
!          - one-slice and bisection moves
!          - Cpu time allocated between different atom groups evenly.
!
!     The external interface of this module consists of:
12 !          - interfaces:
!               - interaction_interface
!          - classes:
!               - AtomGroup
!               - interaction
17 !          - subroutines / functions:
!               - pimc_system_init
!               - relax_system
!               - get_r
!
22 !     To use this module first call "pimc_system_init" and then "relax_system"
!     repeatedly.
!
!     About organization of source code:
!         - Source code is divided into sections with a specified "LEVEL" so
27 !         that higher level subroutines call lower level subroutines.
!



32 module pimc_system
    !use :: performance
    use :: randomgen
    implicit none
    public :: interaction_interface
37  public :: AtomGroup, interaction
    public :: pimc_system_init, relax_system, get_r

    private :: calculate_cpu_times, ag_optimize_dx_max, ag_try_trial_move, &
        ag_one_slice_move, ag_one_slice_move_for_trotter_ne_1, &
42      ag_one_slice_move_for_trotter_eq_1, ag_bisection_move, eval_pot_helper,&
        trindex, pow2below

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! INTERFACES
47 !

    interface
        ! interaction_interface:
        !      - The functions for evaluating potential energy differences must
52      !      be given in this form.
        !      - The function should return potential energy difference generated
        !      when a specific bead is moved to a new location.
        !      - Takes arguments:
        !           group_index: the place of atom group in the "groups_" argument
57      !               of "pimc_system_init".
        !           atom_index : Index of the atom in "r_values" array.
        !           slice_index: Index defining the time slice at which the move
        !               takes place.
        !      - The function should use the "get_r"-function to get the current
62      !      place of the bead in the array as in
        !      "get_r(group_index, atom_index, slice_index)". Also position of
        !      the other atoms should be retrieved in the same way when needed.
        !      - The function should return the energy difference unscaled
        !       (i.e. not divided by the trotter number).
67      !
        function interaction_interface(group_index, atom_index, slice_index, r_new)
            double precision :: interaction_interface
            integer, intent (in) :: group_index, atom_index, slice_index
```

```
            double precision, dimension(3), intent (in) :: r_new
        end function interaction_interface
    end interface

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! CLASSES
!

    type AtomGroup
        ! name:
        !     Name of the group.
        character(len=40) :: name

        ! trotter:
        !     Trotter number for group, must be of form 2^n.
        integer :: trotter

        ! mass
        double precision :: mass

        ! nat:
        !     Number of atoms in group
        ! r_values:
        !     Pointer to atom positions, first index selects the coordinate
        !     (x, y, z). Second index is the "bead_index" that selects
        !     time slice. Third index is the "atom_index" that selects one atom
        !     from the group.
        integer :: nat
        double precision, dimension(:,:,:), pointer :: r_values

        ! nint:
        !     Number of interactions acting on this AtomGroup.
        ! interactions:
        !     Array of indices to the array of interactions passed to
        !     pimc_system_init. Defines to which interactions the group is
        !     subject to.
        integer :: nint
        integer, dimension(:), allocatable :: interactions

        ! dx_max:
        !     Trial move size. This should be set to a sensible (preferably
        !     too small than too big) initial value by the user, but it will
        !     be optimized during initialization.
        double precision :: dx_max

        ! Below is other fields and methods of AtomGroup that are part of
        ! internal implementation of this module.
        double precision :: bead_constant
        integer :: chainl_2pow
        double precision :: cpu_time
        integer :: index
        logical :: use_free_particle_sampling
        integer :: accepted, rejected
    contains
        procedure :: optimize_dx_max => ag_optimize_dx_max
        procedure :: try_trial_move => ag_try_trial_move
        procedure :: one_slice_move => ag_one_slice_move
        procedure :: one_slice_move_for_trotter_ne_1 => ag_one_slice_move_for_trotter_ne_1
        procedure :: one_slice_move_for_trotter_eq_1 => ag_one_slice_move_for_trotter_eq_1
        procedure :: bisection_move => ag_bisection_move
        procedure :: bisection_move_fps => ag_bisection_move_fps
        procedure :: optimize => ag_optimize
        procedure :: get_acceptance_ratio => ag_get_acceptance_ratio
    end type AtomGroup

    !
    type interaction
        ! Potential energy evaluator. More documentation at the definition of
        ! the interface "interaction_interface".
        procedure (interaction_interface), pointer, nopass :: eval

        ! largest_trotter:
        !     The largest trotter number among the atom groups that are subject
        !     to this interaction. This is set automatically.
        integer :: largest_trotter
    end type interaction

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! VARIABLES AND CONSTANTS
!

    integer, private :: relax_rounds
    double precision, private :: beta
```

```
    integer, private :: max_trotter, ngroups
    type (AtomGroup), dimension(:), allocatable, private :: groups
    type (interaction), dimension(:), allocatable, private :: interactions

contains


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! FRONTEND (LEVEL 2)
!

    function get_r(group_index, atom_index, slice_index)
        double precision, dimension(3) :: get_r
        integer, intent (in) :: group_index, atom_index, slice_index
        integer :: trotter_mismatch
        integer :: divee, modee
        double precision :: rem

        trotter_mismatch = max_trotter / groups(group_index)%trotter

        divee = (slice_index - 1) / trotter_mismatch
        modee = mod(slice_index - 1, trotter_mismatch)
        if (modee == 0) then
            get_r = groups(group_index)%r_values(:, divee + 1, atom_index)
            return
        end if
        rem = dble(modee) / trotter_mismatch
        if (divee .NE. groups(group_index)%trotter - 1) then
            get_r = (1d0 - rem) * groups(group_index)%r_values(:, divee + 1, atom_index) &
                + rem * groups(group_index)%r_values(:, divee + 2, atom_index)
            return
        end if

        get_r = (1d0 - rem) * groups(group_index)%r_values(:, divee + 1, atom_index) &
            + rem * groups(group_index)%r_values(:, 1, atom_index)
        return

    end function get_r

    ! pimc_system_init:
    !     This method initializes the simulation.
    !     Arguments:
    !         beta_: The inverse temperature of system
    !         ngroups_: Number of atom groups in the system
    !         groups_: array of instances of "AtomGroup" datastructure
    !         nint_: number of interactions between the atoms
    !         interactions_: array of instances of "interaction" datastructure
    !         time_per_datapoint: amount of time (in seconds) that will be used
    !              for collecting each datapoint
    subroutine pimc_system_init(beta_, ngroups_, groups_, nint_, interactions_, time_per_datapoint)
        double precision :: beta_, total_inverse_cpu_time
        integer :: ngroups_, nint_
        double precision :: time_per_datapoint
        type (AtomGroup), dimension(ngroups_) :: groups_
        type (interaction), dimension(nint_) :: interactions_
        integer :: i, j, k

        ! Set constants
        ngroups = ngroups_
        beta = beta_
        max_trotter = groups_(1)%trotter
        do i = 2, ngroups
            if (max_trotter < groups_(i)%trotter) then
                max_trotter = groups_(i)%trotter
            end if
        end do

        ! Allocate memory
        allocate(groups(ngroups_))
        groups = groups_
        allocate(interactions(nint_))
        interactions = interactions_

        ! Initialize trotter numbers for interactions
        do i = 1, nint_
            interactions(i)%largest_trotter = 1
            do j = 1, ngroups_
                do k = 1, groups(j)%nint
                    if (groups(j)%interactions(k) == i) then
                        if (interactions(i)%largest_trotter < groups(j)%trotter) then
                            interactions(i)%largest_trotter = groups(j)%trotter
                        end if
                    end if
```

```fortran
                end do
            end do
        end do

        ! Calculate bead_constants
        do i = 1, ngroups
            groups(i)%bead_constant = groups(i)%trotter * groups(i)%mass / (2 * beta ** 2)
        end do

        ! init group indices
        do i = 1, ngroups_
            groups(i)%index = i
        end do

        ! Optimization
        print *, "␣␣␣␣Optimizing␣parameters␣for␣fast␣convergence..."
        do i = 1, ngroups
            do j = 1, 3
                call groups(i)%optimize()
            end do
        end do

        ! Calculate cpu time
        print *, "␣␣␣␣Calculating␣cpu␣times␣of␣trial␣moves..."
        call calculate_cpu_times()

        ! Set relax rounds
        total_inverse_cpu_time = 0d0
        do i = 1, ngroups
            total_inverse_cpu_time = total_inverse_cpu_time + 1 / groups(i)%cpu_time
        end do
        relax_rounds = int(time_per_datapoint * total_inverse_cpu_time / ngroups)

        ! Print
        print *, "␣␣␣␣Results:"
        print *, "␣␣␣␣␣␣␣␣steps␣per␣datapoint:␣", relax_rounds
        do i = 1, ngroups
            print *, "␣␣␣␣␣␣␣␣Group:␣", groups(i)%name
            print *, "␣␣␣␣␣␣␣␣␣␣␣␣cpu_time␣per␣trial␣move:␣", groups(i)%cpu_time * 1d9, "ns"
            print *, "␣␣␣␣␣␣␣␣␣␣␣␣number␣of␣trial␣moves␣per␣datapoint:␣",  &
                relax_rounds / total_inverse_cpu_time / groups(i)%cpu_time

            if (groups(i)%trotter .EQ. 1) then
                print *, "␣␣␣␣␣␣␣␣␣␣␣␣uses␣single␣slice␣moves:"
                print *, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣dx_max:␣", groups(i)%dx_max
            else if (groups(i)%use_free_particle_sampling) then
                print *, "␣␣␣␣␣␣␣␣␣␣␣␣uses␣free␣particle␣sampling:"
                print *, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣chainlength:␣", 2 ** groups(i)%chainl_2pow
                print *, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣acceptance:␣", groups(i)%get_acceptance_ratio()
            else
                print *, "␣␣␣␣␣␣␣␣␣␣␣␣uses␣displacement␣moves:"
                print *, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣chainlength:␣", 2 ** groups(i)%chainl_2pow
                print *, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣dx_max:␣", groups(i)%dx_max
            end if
        end do
    end subroutine pimc_system_init

    subroutine relax_system()
        integer :: i
        double precision, dimension(ngroups) :: weights

        do i = 1, ngroups
            weights(i) = 1 / groups(i)%cpu_time
        end do

        weights = weights / sum(weights)

        do i = 1, relax_rounds
            call groups(random_choice(weights, ngroups))%try_trial_move()
        end do

    end subroutine relax_system

    subroutine print_info()
        integer :: i
        print *, "␣␣␣␣␣Final␣acceptance␣ratios:"
        do i = 1, ngroups
            print *, "␣␣␣␣␣␣␣␣Group:␣", groups(i)%name, ";␣", groups(i)%get_acceptance_ratio()
        end do

    end subroutine print_info

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
! SYSTEM METHODS (LEVEL 1)
!     This level and levels below this are not part of external interface.

    subroutine calculate_cpu_times()
        integer :: i, j
        integer :: rounds
        double precision, parameter :: LIMIT = 0.3
        double precision :: t1, t2

        do i = 1, ngroups
            rounds = 10000
            do
                call cpu_time(t1)
                do j = 1, rounds
                    call groups(i)%try_trial_move()
                end do
                call cpu_time(t2)
                if ((t2 - t1) > LIMIT) then
                    exit
                end if
                rounds = 2 * rounds
            end do
            groups(i)%cpu_time = (t2 - t1) / rounds
        end do
    end subroutine calculate_cpu_times


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! GROUP METHODS (LEVEL 0)
!     Contains methods:
!         - ag_optimize_dx_max: This optimizes dx_max of a group
!         - ag_try_trial_move: This runs one simulation step on a group
!

    subroutine ag_optimize(this)
        class (AtomGroup) :: this
        integer, parameter :: TRIAL_MOVES = 100000
        !double precision, parameter :: TARGET_AR = 0.5
        double precision, parameter :: MINIMUM_AR = 0.01
        integer :: max_chainl_2pow, i

        ! If trotter number is one then use one slice moves are used so
        ! only dx_max need to be optimized
        if (this%trotter .EQ. 1) then
            call this%optimize_dx_max()
            return
        end if

        ! It appears that maximal always better at T > 100K so we'll just
        ! use maximum chain lenght always. At temperatures T < 100K this
        ! is probably not that good choice.
        max_chainl_2pow = pow2below(this%trotter)
        this%use_free_particle_sampling = .TRUE.
        this%chainl_2pow = max_chainl_2pow

        ! Calculate acceptance ratio with the selected free particle sampling parameters
        this%accepted = 0
        this%rejected = 0
        do i = 1, TRIAL_MOVES
            call this%try_trial_move()
        end do

        ! If the acceptance ratio is too small don't use free particle sampling
        if (this%get_acceptance_ratio() < MINIMUM_AR) then
            this%use_free_particle_sampling = .FALSE.

            ! Select maximal chain lenght
            this%chainl_2pow = max_chainl_2pow

            ! Optimize dx_max
            call this%optimize_dx_max()
        end if

    end subroutine ag_optimize

    !
    subroutine ag_optimize_dx_max(this)
        class (AtomGroup) :: this
        integer, parameter :: START_OPT_LEVEL = 4
        integer, parameter :: END_OPT_LEVEL = 8
        double precision, parameter :: ACCEPTANCE_LOW = 0.47
        double precision, parameter :: ACCEPTANCE_HIGH = 0.50
        integer, parameter :: BALANCE_DATAPOINTS = 256
```

```
        double precision, parameter :: BALANCE_LIMIT = 0.01
        double precision, parameter :: MULTIPLIER = 1.01
        integer :: i, j, k
        integer :: trial_moves_per_adjustment
        integer :: decrease_counter, increase_counter
        double precision :: ar

        !print *, "start optimize_dx_max", group%index
        trial_moves_per_adjustment = 2 ** START_OPT_LEVEL
        do i = 1, END_OPT_LEVEL - START_OPT_LEVEL
            do
                decrease_counter = 0
                increase_counter = 0
                do j = 1, BALANCE_DATAPOINTS
                    this%accepted = 0
                    this%rejected = 0
                    do k = 1, trial_moves_per_adjustment
                        call this%try_trial_move()
                    end do
                    ar = this%get_acceptance_ratio()
                    if (ar > ACCEPTANCE_HIGH) then
                        increase_counter = increase_counter + 1
                        this%dx_max = this%dx_max * MULTIPLIER
                    else if (ar < ACCEPTANCE_LOW) then
                        decrease_counter = decrease_counter + 1
                        this%dx_max = this%dx_max / MULTIPLIER
                    end if
                end do
                !print *, increase_counter, decrease_counter, group%dx_max
                if (abs(decrease_counter - increase_counter) < BALANCE_LIMIT &
                    * BALANCE_DATAPOINTS) then
                    exit
                end if
            end do
            trial_moves_per_adjustment = 2 * trial_moves_per_adjustment
        end do
        !print *, "end optimize_dx_max"
    end subroutine ag_optimize_dx_max

    subroutine ag_try_trial_move(this)
        class (AtomGroup) :: this
        if (this%trotter .EQ. 1) then
            call this%one_slice_move_for_trotter_eq_1()
        else
            if (this%use_free_particle_sampling) then
                call this%bisection_move_fps()
            else
                call this%bisection_move()
            end if
        end if
    end subroutine ag_try_trial_move


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! GROUP METHODS; (LEVEL -1)
!     Contains methods:
!         - ag_one_slice_move: make an one slice type move on a group
!         - ag_bisection_move: make a bisection move
!

    function ag_get_acceptance_ratio(this)
        class (AtomGroup) :: this
        double precision :: ag_get_acceptance_ratio
        ag_get_acceptance_ratio = dble(this%accepted) / dble(this%accepted + this%rejected)
        return
    end function ag_get_acceptance_ratio

    ! DEPRECATED
    subroutine ag_one_slice_move(this)
        class (AtomGroup) :: this
        if (this%trotter .EQ. 1) then
            call this%one_slice_move_for_trotter_eq_1()
        else
            call this%one_slice_move_for_trotter_ne_1()
        end if
    end subroutine ag_one_slice_move


    ! Only for case this%trotter > 1
    subroutine ag_bisection_move(this)
        class (AtomGroup) :: this
        integer :: chainl, atom_index, bead1_ind
        double precision, dimension(3, 2 ** this%chainl_2pow + 1) :: r_values_new, displacement
```

```
        integer :: first, stride, halfstride
        double precision :: sigma, sqsum, ed
        integer :: i, j

        !print *, "Begin bis_move"
        atom_index = random_int(this%nat)
        bead1_ind = random_int(this%trotter)
        chainl = 2 ** this%chainl_2pow

        ! Generate displacement using bisection method
        displacement(:, 1) = 0d0
        displacement(:, chainl + 1) = 0d0
        do i = 0, this%chainl_2pow - 1
            first = 2 ** (this%chainl_2pow - i - 1) + 1
            stride = 2 ** (this%chainl_2pow - i)
            halfstride = stride / 2
            sigma = this%dx_max * sqrt(dble(2 ** (this%chainl_2pow - i - 1)))
            do j = first, chainl, stride
                displacement(:, j) = 0.5 * ( &
                      displacement(:, j - halfstride) &
                    + displacement(:, j + halfstride) &
                ) + random_normal_3d(sigma)
            end do
        end do

        ! Calculate new suggested positions
        do i = 1, chainl + 1
            r_values_new(:, i) = this%r_values(:, trindex(bead1_ind + i - 1, &
                this%trotter), atom_index) + displacement(:, i)
        end do

        ! Evaluate move; bead-bead interaction
        sqsum = 0
        do i = 1, chainl
            sqsum = sqsum + sum((r_values_new(:, i) - r_values_new(:, i + 1)) ** 2) &
                - sum( &
                    (this%r_values(:, trindex(bead1_ind + i - 1, this%trotter), atom_index) &
                    - this%r_values(:, trindex(bead1_ind + i, this%trotter), atom_index)) ** 2 &
                )
        end do
        ed = sqsum * this%bead_constant

        ! Evaluate move; potential energy part
        do i = 1, chainl
            ed = ed + eval_pot_helper(this, atom_index, trindex(bead1_ind + i - 1, this%trotter) &
                , r_values_new(:, i), r_values_new(:, i + 1))
        end do

        ! Accept or reject move
        if (ed < 0 .OR. exp(-beta * ed) >= random_float(0.0d0, 1.0d0)) then
            do i = 2, chainl
                this%r_values(:, trindex(bead1_ind + i - 1, this%trotter), atom_index) = &
                    r_values_new(:, i)
            end do
            this%accepted = this%accepted + 1
        else
            this%rejected = this%rejected + 1
        end if
    end subroutine ag_bisection_move


    ! Only for case this%trotter > 1
    subroutine ag_bisection_move_fps(this)
        class (AtomGroup) :: this
        integer :: chainl, atom_index, bead1_ind
        double precision, dimension(3, 2 ** this%chainl_2pow + 1) :: r_values_new
        integer :: first, stride, halfstride
        double precision :: sigma, ed
        integer :: i, j

        !print *, "Begin bis_move"
        atom_index = random_int(this%nat)
        bead1_ind = random_int(this%trotter)
        chainl = 2 ** this%chainl_2pow

        ! Generate displacement using bisection method
        r_values_new(:, 1) = this%r_values(:, trindex(bead1_ind, this%trotter), atom_index)
        r_values_new(:, chainl + 1) = this%r_values(:, trindex(bead1_ind + chainl, this%trotter), atom_index)
        do i = 0, this%chainl_2pow - 1
            first = 2 ** (this%chainl_2pow - i - 1) + 1
            stride = 2 ** (this%chainl_2pow - i)
            halfstride = stride / 2
            !sigma = this%dx_max * sqrt(dble(2 ** (this%chainl_2pow - i - 1)))
```

```
            sigma = sqrt((beta * 2 ** (this%chainl_2pow - i - 1)) / (2 * this%mass * this%trotter))
            do j = first, chainl, stride
                r_values_new(:, j) = 0.5 * ( &
                      r_values_new(:, j - halfstride) &
                    + r_values_new(:, j + halfstride) &
                ) + random_normal_3d(sigma)
            end do
        end do

        ! Evaluate move; potential energy part
        ed = 0d0
        do i = 1, chainl
            ed = ed + eval_pot_helper(this, atom_index, trindex(bead1_ind + i - 1, this%trotter) &
                , r_values_new(:, i), r_values_new(:, i + 1))
        end do

        ! Accept or reject move
        if (ed < 0 .OR. exp(-beta * ed) >= random_float(0.0d0, 1.0d0)) then
            do i = 2, chainl
                this%r_values(:, trindex(bead1_ind + i - 1, this%trotter), atom_index) = &
                    r_values_new(:, i)
            end do
            this%accepted = this%accepted + 1
        else
            this%rejected = this%rejected + 1
        end if
    end subroutine ag_bisection_move_fps

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! HELPING FUNCTIONS 1 (LEVEL -2)
!     Contains methods:
!          - eval_pot_helper: function used by ag_bisection_move
!          - ag_one_slice_move_for_trotter_ne_1: used by ag_one_slice_move
!          - ag_one_slice_move_for_trotter_eq_1: used by ag_one_slice_move
!

    ! If bead1 = group%r_values(:, bead_index, atom_index) and
    ! bead2 = group%r_values(:, bead_index + 1, atom_index), then what is the
    ! potential energy difference generated by bead1 and all the "ghost" beads
    ! between bead1 and bead2 (but not including bead2), when bead1 moves to r_new1
    ! and bead2 moves to r_new2?
    function eval_pot_helper(group, atom_index, bead_index, r_new1, r_new2)
        double precision :: eval_pot_helper
        type (AtomGroup) :: group
        integer :: atom_index, bead_index, stride1, stride2, int_index
        integer :: trotter_int, i, j, index, slice_index
        double precision, dimension(3) :: r_new1, r_new2, r
        double precision :: temp, ed, factor

        ed = 0d0
        do i = 1, group%nint
            int_index = group%interactions(i)
            trotter_int = interactions(int_index)%largest_trotter
            stride1 = max_trotter / trotter_int
            stride2 = trotter_int / group%trotter
            temp = 0d0
            do j = 0, stride2 - 1
                factor = dble(j) / stride2
                r = (1d0 - factor) * r_new1 + factor * r_new2
                index = trindex((bead_index - 1) * stride2 + 1 + j, trotter_int)
                slice_index = (index - 1) * stride1 + 1
                temp = temp + interactions(int_index)%eval(group%index, atom_index, slice_index, r)
            end do
            ed = ed + temp / trotter_int
        end do
        eval_pot_helper = ed
        return
    end function eval_pot_helper

    ! Only for case group%trotter > 1
    ! DEPRECATED
    subroutine ag_one_slice_move_for_trotter_ne_1(this)
        class (AtomGroup) :: this
        double precision, dimension(3) :: r_new, r_cur, r1, r2, r
        integer :: atom_index, bead_index
        double precision :: ed
        integer :: stride1, stride2, slice_index, index
        double precision :: factor, temp
        integer :: i, j, int_index
        integer :: trotter_int

        !print *, "Begin osl1_move"
```

```fortran
        atom_index = random_int(this%nat)
        bead_index = random_int(this%trotter)

        r_cur = this%r_values(:, bead_index, atom_index)
        r_new = r_cur + random_normal_3d(this%dx_max)

        r1 = this%r_values(:, trindex(bead_index - 1, this%trotter), atom_index)
        r2 = this%r_values(:, trindex(bead_index + 1, this%trotter), atom_index)

        ed = 0d0
        ! Evaluate potential energy
        do i = 1, this%nint
            int_index = this%interactions(i)
            trotter_int = interactions(int_index)%largest_trotter
            stride1 = max_trotter / trotter_int
            stride2 = trotter_int / this%trotter
            temp = 0d0
            do j = -stride2 + 1, -1
                factor = dble(abs(j)) / stride2
                r = r1 * factor + r_new * (1d0 - factor)
                index = trindex((bead_index - 1) * stride2 + 1 + j, trotter_int)
                slice_index = (index - 1) * stride1 + 1
                temp = temp + interactions(int_index)%eval(this%index, atom_index, slice_index, r)
            end do
            do j = 0, stride2 - 1
                factor = dble(j) / stride2
                r = r2 * factor + r_new * (1d0 - factor)
                index = trindex((bead_index - 1) * stride2 + 1 + j, trotter_int)
                slice_index = (index - 1) * stride1 + 1
                temp = temp + interactions(int_index)%eval(this%index, atom_index, slice_index, r)
            end do
            ed = ed + temp / trotter_int
        end do

        ! Evaluate bead energy
        ed = ed + this%bead_constant * (sum((r_new - r1) ** 2) + sum((r_new - r2) ** 2) &
             - sum((r_cur - r1) ** 2) - sum((r_cur - r2) ** 2))

        ! Accept or reject the trial move
        if (ed < 0 .OR. exp(-beta * ed) >= random_float(0.0d0, 1.0d0)) then
            this%r_values(:, bead_index, atom_index) = r_new
            this%accepted = this%accepted + 1
        else
            this%rejected = this%rejected + 1
        end if
    end subroutine ag_one_slice_move_for_trotter_ne_1

    ! Only for case this%trotter = 1
    subroutine ag_one_slice_move_for_trotter_eq_1(this)
        class (AtomGroup) :: this
        integer :: atom_index, int_index, trotter_int, stride, i, j
        integer, parameter :: BEAD_INDEX = 1
        double precision :: ed, temp
        double precision, dimension(3) :: r_new

        atom_index = random_int(this%nat)
        r_new = this%r_values(:, BEAD_INDEX, atom_index) + random_normal_3d(this%dx_max)
        ed = 0d0
        do i = 1, this%nint
            int_index = this%interactions(i)
            trotter_int = interactions(int_index)%largest_trotter
            stride = max_trotter / trotter_int
            temp = 0d0
            do j = 1, max_trotter, stride
                temp = temp + interactions(int_index)%eval(this%index, atom_index, j, r_new)
            end do
            ed = ed + temp / trotter_int
        end do

        ! Accept or reject the trial move
        if (ed < 0 .OR. exp(-beta * ed) >= random_float(0.0d0, 1.0d0)) then
            this%r_values(:, BEAD_INDEX, atom_index) = r_new
            this%accepted = this%accepted + 1
        else
            this%rejected = this%rejected + 1
        end if
    end subroutine ag_one_slice_move_for_trotter_eq_1


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! HELPING FUNCTIONS 2 (LEVEL -3)
!     Other useful functions
```

```
    ! Translate index
    pure function trindex(ind, base)
        integer :: trindex
        integer, intent (in) :: ind, base
        trindex = modulo(ind - 1, base) + 1
        return
    end function trindex

    ! Calculates largest integer k_2pow such that 2^k_2pow <= n
    pure function pow2below(n)
        integer :: pow2below
        integer, intent (in) :: n
        integer :: k, k_2pow

        k_2pow = 0
        k = 1
        do
            if (k > n) then
                pow2below = k_2pow - 1
                return
            end if
            k_2pow = k_2pow + 1
            k = 2 * k
        end do
    end function pow2below

end module pimc_system
```

Contents of file constants.f90:

```
module constants
    implicit none


    ! Dalton to electron mass conversion
    double precision, parameter, private :: DALTON_TO_ME = 1822.888486192d0

    ! Mass of Helium atoms in Daltons
    double precision, parameter, private :: MASS_HE3 = 3.0160293d0
    double precision, parameter, public :: MASS_HE3_ATOMIC = MASS_HE3 * DALTON_TO_ME

    ! Mass of carbon atoms in Daltons
    double precision, parameter, private :: MASS_C = 12.0d0
    double precision, parameter, public :: MASS_C_ATOMIC = MASS_C * DALTON_TO_ME


    ! Angstrom to Bohr radius conversion
    double precision, parameter, public :: ANGS_TO_BOHR = 1.8897261254578281d0

    ! Kelvin to Hartree conversion; based on Boltzmann constant
    double precision, parameter, public :: KELVIN_TO_HARTREE = 3.166810514923162d-06

    ! The array C_NEIGHBOURS(:, j) contains atom indices of the neighbours and
    ! neighbours of neighbours of the carbon atom with index j.
    ! If the array C_NEIGHBOURS(:, j) is (1,2,3,4,5,6,7,8,9), and a hh-bond
    ! between atoms x and y is denoted by hh(x,y), then there are the following
    ! bonds:
    ! hh(j, 1), ph(j, 2), ph(j, 3),
    ! ph(1, 4), ph(1, 5), hh(2, 6),
    ! ph(2, 7), hh(3, 8), ph(3, 9).
    ! This array was implicitly inferred from FULLERENE_R_VALUES.
    integer, parameter, dimension (9, 60), public :: C_NEIGHBOURS = reshape((/&
        24,  2,  5, 23, 25, 29,  3, 19,  4,&!1
        29,  1,  3, 28, 30, 24,  5,  9,  4,&!2
         9,  2,  4,  8, 10, 29,  1, 14,  5,&!3
        14,  3,  5, 13, 15,  9,  2, 19,  1,&!4
        19,  1,  4, 18, 20, 24,  2, 14,  3,&!5
        34,  7, 10, 33, 35, 40,  8, 28,  9,&!6
        40,  6,  8, 36, 39, 34, 10, 15,  9,&!7
        15,  7,  9, 11, 14, 40,  6,  3, 10,&!8
         3,  8, 10,  2,  4, 15,  7, 28,  6,&!9
        28,  6,  9, 27, 29, 34,  7,  3,  8,&!10
        39, 12, 15, 38, 40, 45, 13,  8, 14,&!11
        45, 11, 13, 41, 44, 39, 15, 20, 14,&!12
        20, 12, 14, 16, 19, 45, 11,  4, 15,&!13
         4, 13, 15,  3,  5, 20, 12,  8, 11,&!14
         8, 11, 14,  7,  9, 39, 12,  4, 13,&!15
        44, 17, 20, 43, 45, 50, 18, 13, 19,&!16
        50, 16, 18, 46, 49, 44, 20, 25, 19,&!17
        25, 17, 19, 21, 24, 50, 16,  5, 20,&!18
         5, 18, 20,  1,  4, 25, 17, 13, 16,&!19
```

```
        13, 16, 19, 12, 14, 44, 17,  5, 18,&!20
        49, 22, 25, 48, 50, 55, 23, 18, 24,&!21
        55, 21, 23, 51, 54, 49, 25, 30, 24,&!22
        30, 22, 24, 26, 29, 55, 21,  1, 25,&!23
         1, 23, 25,  2,  5, 30, 22, 18, 21,&!24
        18, 21, 24, 17, 19, 49, 22,  1, 23,&!25
        54, 27, 30, 53, 55, 35, 28, 23, 29,&!26
        35, 26, 28, 31, 34, 54, 30, 10, 29,&!27
        10, 27, 29,  6,  9, 35, 26,  2, 30,&!28
         2, 28, 30,  1,  3, 10, 27, 23, 26,&!29
        23, 26, 29, 22, 24, 54, 27,  2, 28,&!30
        53, 32, 35, 52, 54, 60, 33, 27, 34,&!31
        60, 31, 33, 56, 59, 53, 35, 36, 34,&!32
        36, 32, 34, 37, 40, 60, 31,  6, 35,&!33
         6, 33, 35,  7, 10, 36, 32, 27, 31,&!34
        27, 31, 34, 26, 28, 53, 32,  6, 33,&!35
        33, 37, 40, 32, 34, 59, 38,  7, 39,&!36
        59, 36, 38, 58, 60, 33, 40, 41, 39,&!37
        41, 37, 39, 42, 45, 59, 36, 11, 40,&!38
        11, 38, 40, 12, 15, 41, 37,  7, 36,&!39
         7, 36, 39,  6,  8, 33, 37, 11, 38,&!40
        38, 42, 45, 37, 39, 58, 43, 12, 44,&!41
        58, 41, 43, 57, 59, 38, 45, 46, 44,&!42
        46, 42, 44, 47, 50, 58, 41, 16, 45,&!43
        16, 43, 45, 17, 20, 46, 42, 12, 41,&!44
        12, 41, 44, 11, 13, 38, 42, 16, 43,&!45
        43, 47, 50, 42, 44, 57, 48, 17, 49,&!46
        57, 46, 48, 56, 58, 43, 50, 51, 49,&!47
        51, 47, 49, 52, 55, 57, 46, 21, 50,&!48
        21, 48, 50, 22, 25, 51, 47, 17, 46,&!49
        17, 46, 49, 16, 18, 43, 47, 21, 48,&!50
        48, 52, 55, 47, 49, 56, 53, 22, 54,&!51
        56, 51, 53, 57, 60, 48, 55, 31, 54,&!52
        31, 52, 54, 32, 35, 56, 51, 26, 55,&!53
        26, 53, 55, 27, 30, 31, 52, 22, 51,&!54
        22, 51, 54, 21, 23, 48, 52, 26, 53,&!55
        52, 57, 60, 51, 53, 47, 58, 32, 59,&!56
        47, 56, 58, 46, 48, 52, 60, 42, 59,&!57
        42, 57, 59, 41, 43, 47, 56, 37, 60,&!58
        37, 58, 60, 36, 38, 42, 57, 32, 56,&!59
        32, 56, 59, 31, 33, 52, 57, 37, 58 &!60
    /), shape(C_NEIGHBOURS))

    ! Structure taken from Straka (3Atomic_fit_He@C60.xlsx)(,,,)
    double precision, parameter, dimension(3, 60), public :: FULLERENE_R_VALUES = reshape((/&
         0.724251, -2.229014,  6.310147,&
         2.343724,  0.000000,  6.310147,&
         0.724251,  2.229014,  6.310147,&
        -1.896113,  1.377606,  6.310147,&
        -1.896113, -1.377606,  6.310147,&
         3.316300,  5.748493,  1.126049,&
         0.695936,  6.599900,  1.126049,&
        -0.475926,  5.748493,  3.469773,&
         1.420187,  4.370886,  4.918274,&
         3.763911,  4.370886,  3.469773,&
        -4.442348,  4.930370,  1.126049,&
        -6.061822,  2.701356,  1.126049,&
        -5.614210,  1.323750,  3.469773,&
        -3.718098,  2.701356,  4.918274,&
        -2.993847,  4.930370,  3.469773,&
        -6.061822, -2.701356,  1.126049,&
        -4.442348, -4.930370,  1.126049,&
        -2.993847, -4.930370,  3.469773,&
        -3.718098, -2.701356,  4.918274,&
        -5.614210, -1.323750,  3.469773,&
         0.695936, -6.599900,  1.126049,&
         3.316300, -5.748493,  1.126049,&
         3.763911, -4.370886,  3.469773,&
         1.420187, -4.370886,  4.918274,&
        -0.475926, -5.748493,  3.469773,&
         6.491934, -1.377606,  1.126049,&
         6.491934,  1.377606,  1.126049,&
         5.320072,  2.229014,  3.469773,&
         4.595822,  0.000000,  4.918274,&
         5.320072, -2.229014,  3.469773,&
         5.614210,  1.323750, -3.469773,&
         3.718098,  2.701356, -4.918274,&
         2.993847,  4.930370, -3.469773,&
         4.442348,  4.930370, -1.126049,&
         6.061822,  2.701356, -1.126049,&
         0.475926,  5.748493, -3.469773,&
        -1.420187,  4.370886, -4.918274,&
        -3.763911,  4.370886, -3.469773,&
```

```
        -3.316300,  5.748493, -1.126049,&
        -0.695936,  6.599900, -1.126049,&
        -5.320072,  2.229014, -3.469773,&
        -4.595822,  0.000000, -4.918274,&
        -5.320072, -2.229014, -3.469773,&
        -6.491934, -1.377606, -1.126049,&
        -6.491934,  1.377606, -1.126049,&
        -3.763911, -4.370886, -3.469773,&
        -1.420187, -4.370886, -4.918274,&
         0.475926, -5.748493, -3.469773,&
        -0.695936, -6.599900, -1.126049,&
        -3.316300, -5.748493, -1.126049,&
         2.993847, -4.930370, -3.469773,&
         3.718098, -2.701356, -4.918274,&
         5.614210, -1.323750, -3.469773,&
         6.061822, -2.701356, -1.126049,&
         4.442348, -4.930370, -1.126049,&
         1.896113, -1.377606, -6.310147,&
        -0.724251, -2.229014, -6.310147,&
        -2.343724,  0.000000, -6.310147,&
        -0.724251,  2.229014, -6.310147,&
         1.896113,  1.377606, -6.310147&
    /), shape(FULLERENE_R_VALUES))

    ! Wu force field from Ceulemans et al.(2001)
    double precision, parameter, private :: CONVERSION_FACTOR = 0.0006423048659340211d0
    double precision, parameter, public :: wu_c1 = 520.1d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c2 = 348.1d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c3 = 68.4d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c4 = 77.6d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c5 = 28.0d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c6 = 9.8d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c7 = 35.1d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c8 = 21.7d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c9 = 42.7d0*CONVERSION_FACTOR
    double precision, parameter, public :: wu_c10 = 11.6d0*CONVERSION_FACTOR

    ! Lengths of bonds
    double precision, parameter, public :: wu_hh_r = 2.647500! Value taken from FULLERENE_R_VALUES implicitly
    double precision, parameter, public :: wu_ph_r = 2.755213! Value taken from FULLERENE_R_VALUES implicitly

    ! Distances of atoms separated by two bonds
    double precision, parameter, public :: wu_h_r = sqrt(wu_hh_r ** 2 + wu_ph_r ** 2 + wu_hh_r * wu_ph_r)
    double precision, parameter, public :: wu_p_r = wu_ph_r * (1d0 + sqrt(5d0)) / 2d0

    ! Angles
    double precision, parameter, public :: wu_p_a = 1.8849555921538759d0
    double precision, parameter, public :: wu_h_a = 2.0943951023931954d0

    ! C70-fullerene structure, generated by modifying the C60-fullerene structure... not optimized
    double precision, parameter, dimension(3, 70), public :: C70_R_VALUES = reshape((/&
        7.24251000d-01, -2.22901400d+00,  7.43619600d+00,&
        2.34372400d+00,  0.00000000d+00,  7.43619600d+00,&
        7.24251000d-01,  2.22901400d+00,  7.43619600d+00,&
       -1.89611300d+00,  1.37760600d+00,  7.43619600d+00,&
       -1.89611300d+00, -1.37760600d+00,  7.43619600d+00,&
        3.31630000d+00,  5.74849300d+00,  2.25209800d+00,&
        6.95936000d-01,  6.59990000d+00,  2.25209800d+00,&
       -4.75926000d-01,  5.74849300d+00,  4.59582200d+00,&
        1.42018700d+00,  4.37088600d+00,  6.04432300d+00,&
        3.76391100d+00,  4.37088600d+00,  4.59582200d+00,&
       -4.44234800d+00,  4.93037000d+00,  2.25209800d+00,&
       -6.06182200d+00,  2.70135600d+00,  2.25209800d+00,&
       -5.61421000d+00,  1.32375000d+00,  4.59582200d+00,&
       -3.71809800d+00,  2.70135600d+00,  6.04432300d+00,&
       -2.99384700d+00,  4.93037000d+00,  4.59582200d+00,&
       -6.06182200d+00, -2.70135600d+00,  2.25209800d+00,&
       -4.44234800d+00, -4.93037000d+00,  2.25209800d+00,&
       -2.99384700d+00, -4.93037000d+00,  4.59582200d+00,&
       -3.71809800d+00, -2.70135600d+00,  6.04432300d+00,&
       -5.61421000d+00, -1.32375000d+00,  4.59582200d+00,&
        6.95936000d-01, -6.59990000d+00,  2.25209800d+00,&
        3.31630000d+00, -5.74849300d+00,  2.25209800d+00,&
        3.76391100d+00, -4.37088600d+00,  4.59582200d+00,&
        1.42018700d+00, -4.37088600d+00,  6.04432300d+00,&
       -4.75926000d-01, -5.74849300d+00,  4.59582200d+00,&
        6.49193400d+00, -1.37760600d+00,  2.25209800d+00,&
        6.49193400d+00,  1.37760600d+00,  2.25209800d+00,&
        5.32007200d+00,  2.22901400d+00,  4.59582200d+00,&
        4.59582200d+00,  0.00000000d+00,  6.04432300d+00,&
        5.32007200d+00, -2.22901400d+00,  4.59582200d+00,&
        6.06182247d+00,  2.70135630d+00,  0.00000000d+00,&
        4.44234794d+00,  4.93037034d+00,  0.00000000d+00,&
```

```
       -6.95936253d-01,  6.59989976d+00,  0.00000000d+00,&
       -3.31629980d+00,  5.74849249d+00,  0.00000000d+00,&
       -6.49193423d+00,  1.37760666d+00,  0.00000000d+00,&
       -6.49193380d+00, -1.37760648d+00,  0.00000000d+00,&
       -3.31629984d+00, -5.74849218d+00,  0.00000000d+00,&
       -6.95936350d-01, -6.59990076d+00,  0.00000000d+00,&
        4.44234844d+00, -4.93036973d+00,  0.00000000d+00,&
        6.06182142d+00, -2.70135640d+00,  0.00000000d+00,&
        5.32007203d+00, -2.22901359d+00, -4.59582200d+00,&
        4.59582169d+00, -2.59121409d-07, -6.04432300d+00,&
        5.32007188d+00,  2.22901400d+00, -4.59582200d+00,&
        6.49193380d+00,  1.37760648d+00, -2.25209800d+00,&
        6.49193423d+00, -1.37760666d+00, -2.25209800d+00,&
        3.76391163d+00,  4.37088625d+00, -4.59582200d+00,&
        1.42018691d+00,  4.37088603d+00, -6.04432300d+00,&
       -4.75925634d-01,  5.74849243d+00, -4.59582200d+00,&
        6.95936350d-01,  6.59990076d+00, -2.25209800d+00,&
        3.31629984d+00,  5.74849218d+00, -2.25209800d+00,&
       -2.99384710d+00,  4.93037007d+00, -4.59582200d+00,&
       -3.71809810d+00,  2.70135639d+00, -6.04432300d+00,&
       -5.61421022d+00,  1.32374966d+00, -4.59582200d+00,&
       -6.06182142d+00,  2.70135640d+00, -2.25209800d+00,&
       -4.44234844d+00,  4.93036973d+00, -2.25209800d+00,&
       -5.61421029d+00, -1.32374968d+00, -4.59582200d+00,&
       -3.71809775d+00, -2.70135608d+00, -6.04432300d+00,&
       -2.99384719d+00, -4.93037081d+00, -4.59582200d+00,&
       -4.44234794d+00, -4.93037034d+00, -2.25209800d+00,&
       -6.06182247d+00, -2.70135630d+00, -2.25209800d+00,&
       -4.75925673d-01, -5.74849223d+00, -4.59582200d+00,&
        1.42018725d+00, -4.37088608d+00, -6.04432300d+00,&
        3.76391057d+00, -4.37088609d+00, -4.59582200d+00,&
        3.31629980d+00, -5.74849249d+00, -2.25209800d+00,&
        6.95936253d-01, -6.59989976d+00, -2.25209800d+00,&
        7.24251150d-01, -2.22901392d+00, -7.43619600d+00,&
       -1.89611292d+00, -1.37760615d+00, -7.43619600d+00,&
       -1.89611255d+00,  1.37760640d+00, -7.43619600d+00,&
        7.24250189d-01,  2.22901426d+00, -7.43619600d+00,&
        2.34372413d+00, -5.92527144d-07, -7.43619600d+00&
    /), shape(C70_R_VALUES))

end module constants
```

Contents of file pair_potential.f90:

```
module pair_potential
    use :: constants
    implicit none

contains

    ! He-He pair potential, from Katja Hyvönen's thesis (2015)
    pure function he_he_potential(r)
        double precision :: he_he_potential, value, factor
        double precision, intent (in) :: r
        factor = ANGS_TO_BOHR / r
        value = 2.37466d7 * factor
        value = (value - 5.30363d7) * factor
        value = (value + 4.37807d7) * factor
        value = (value - 1.62169d7) * factor
        value = (value + 2.75932d6) * factor
        he_he_potential = ((value - 183915.0d0) * factor ** 6) * KELVIN_TO_HARTREE
        return
    end function he_he_potential

    ! Straka (3Atomic_fit_He@C60.xlsx)(,,,)
    pure function he_c_potential(r)
        double precision, intent (in) :: r
        double precision :: he_c_potential, value, factor
        factor = ANGS_TO_BOHR / r
        value = 45930262.3536588d0 * factor
        value = (value + 45759710.5441796d0) * factor
        value = (value + 28479845.9140703d0) * factor
        value = (value - 8323743.5809365d0) * factor
        value = (value - 44211024.3257504d0) * factor
        value = (value - 27379844.6972154d0) * factor
        value = (value + 54959330.4400239d0) * factor
        value = (value - 20855618.0087883d0) * factor
        he_c_potential = ((value + 2359205.20565829d0) * factor ** 6) * KELVIN_TO_HARTREE
        return
    end function he_c_potential
```

```
    ! (,,,)
    pure function he_c_potential_new_neutral(r)
        double precision, intent (in) :: r
        double precision, parameter :: c6 = 247780.4382
        double precision, parameter :: c8 = -9141711.461
        double precision, parameter :: c10 = 100344403.9
        double precision, parameter :: c12 = -415991262.2
        double precision, parameter :: c14 = 635716691
        double precision :: he_c_potential_new_neutral, factor

        factor = (ANGS_TO_BOHR / r) ** 2
        he_c_potential_new_neutral = KELVIN_TO_HARTREE * (factor ** 3) * ( &
            c6 + factor * ( &
            c8 + factor * ( &
            c10 + factor * ( &
            c12 + factor * c14))))
        return
    end function he_c_potential_new_neutral

    ! deprecated
!    pure function he_c_potential_new_anion(r)
!        double precision, intent (in) :: r
!        double precision, parameter :: c6 = 125123.4183
!        double precision, parameter :: c8 = -5543434.474
!        double precision, parameter :: c10 = 61018478.94
!        double precision, parameter :: c12 = -231960973
!        double precision, parameter :: c14 = 318690312.9
!        double precision :: he_c_potential_new_anion, factor
!
!        factor = (ANGS_TO_BOHR / r) ** 2
!        he_c_potential_new_anion =  KELVIN_TO_HARTREE * (factor ** 3) * ( &
!            c6 + factor * ( &
!            c8 + factor * ( &
!            c10 + factor * ( &
!            c12 + factor * c14))))
!        return
!
!    end function he_c_potential_new_anion

    ! (,,,)
    pure function he_c_potential_new_anion(r)
        double precision, intent (in) :: r
        double precision, parameter :: c6 = 186093.848
        double precision, parameter :: c8 = -7472700.951
        double precision, parameter :: c10 = 82958587.52
        double precision, parameter :: c12 = -339581069.7
        double precision, parameter :: c14 = 511791498.8
        double precision :: he_c_potential_new_anion, factor

        factor = (ANGS_TO_BOHR / r) ** 2
        he_c_potential_new_anion =  KELVIN_TO_HARTREE * (factor ** 3) * ( &
            c6 + factor * ( &
            c8 + factor * ( &
            c10 + factor * ( &
            c12 + factor * c14))))
        return

    end function he_c_potential_new_anion
    ! Effective spherical potential from Katja Hyvönen's thesis (2015)
    pure function fullerene_potential(r)
        double precision :: fullerene_potential, value, factor
        double precision, dimension(3), intent (in) :: r
        factor = sum((r / ANGS_TO_BOHR) ** 2)
        value = 366.918 * factor
        value = (value + 259.816) * factor
        fullerene_potential = ((value + 397.763) * factor) * KELVIN_TO_HARTREE
        return
    end function fullerene_potential

end module
```

Contents of file randomgen.f90:

```
module randomgen
    use, intrinsic :: ISO_Fortran_env
    implicit none

    public :: random_float, random_int
    public :: random_normal_3d

    private :: gauss, marsaglia_pm
```

```
contains

    ! Random number generator for floats
    function random_float(start, end)
        double precision, intent (in) :: start, end
        double precision :: random_float
        call random_number(random_float)
        random_float = start + (end - start) * random_float
        return
    end function random_float

    ! Random number generator for integers
    function random_int(n)
        integer :: random_int
        integer, intent (in) :: n
        double precision :: r
        call random_number(r)
        random_int = int(n * r) + 1

        ! To be certain...
        if (random_int .GT. n) then
            random_int = n
        end if

        return
    end function random_int

    ! gaussian distribution function
    pure function gauss(sigma, x)
        double precision, intent (in) :: sigma, x
        double precision :: gauss
        double precision, parameter :: PI = 3.14159265358979323846264338d0
        gauss = exp(-0.5 * (x / sigma) ** 2) / sqrt(2 * PI * sigma ** 2)
        return
    end function gauss


    ! Normal distribution generator, (Marsaglia polar method from Wikipedia)
    subroutine marsaglia_pm(rn1, rn2)
        double precision :: rn1, rn2, x, y, s, temp

        do
            x = random_float(-1d0, 1d0)
            y = random_float(-1d0, 1d0)
            s = x ** 2 + y ** 2
            if (s < 1) then
                temp = sqrt(-2 * log(s) / s)
                rn1 = x * temp
                rn2 = y * temp
                return
            end if
        end do

    end subroutine marsaglia_pm


    ! 3D normal distribution
    function random_normal_3d(sigma)
        double precision, intent (in) :: sigma
        double precision, dimension(3) :: random_normal_3d, result
        call marsaglia_pm(result(1), result(2))
        call marsaglia_pm(result(2), result(3))
        random_normal_3d = sigma * result
        return
    end function random_normal_3d


    ! Randomly select an integer between 1,..., length, with probability
    ! proportional to weights. (rejection method)
    function random_choice(weights, length)
        integer :: random_choice
        integer :: length
        double precision, dimension(length) :: weights

        do
            random_choice = random_int(length)
            if (weights(random_choice) > random_float(0d0, 1d0)) then
                exit
            end if
        end do
        return
    end function random_choice
```

```
93
end module randomgen
```

B.2 Postprocessor

Contents of file postproc.cpp:

```
1  /*
       Supported simulation types:


       RIGID - will use the best shift functions for neutral case
6      SPHERICAL - will use the best shift functions for neutral case
       FULL_NEW_NEUTRAL - will use the best shift functions for neutral case
       FULL_NEW_ANION - will use the best shift functions for anion case



11
   */

   #include <iostream>
   #include <string.h>
16 #include <inttypes.h>
   #include <ios>
   #include <fstream>
   #include <cmath>
   #include <set>
21
   #include "distance.cpp"
   #include "chem_shift.cpp"
   #include "constants.cpp"

26



   #define FULL_SIM  0
   #define HE_ONLY_SIM 1
   #define CAVITY_SIM 2
31 #define C70_RIGID_SIM 3
   #define FULL_NEW_NEUTRAL 4
   #define FULL_NEW_ANION 5

   #define BUFFER_SIZE 4096
36 #define VERSION_NAME_LENGTH 16


   char input_buffer[BUFFER_SIZE];
   char filename_prefix[BUFFER_SIZE];
41 char filename[BUFFER_SIZE];

   double he_r_values[2][3];
   double c_r_values[C_ATOMS_C60][3];

46

   const std::set<int> supported_simulations = {
       HE_ONLY_SIM,
       CAVITY_SIM,
       FULL_NEW_NEUTRAL,
51     FULL_NEW_ANION
   };

   const std::set<int> full_simulations = {
       FULL_SIM,
56     FULL_NEW_NEUTRAL,
       FULL_NEW_ANION
   };

   bool contains(std::set<int> set1, int key) {
61     if (set1.find(key) != set1.end()) {
           return true;
       }
       return false;
   }
66
   void move_to_center_of_mass(
       double he_r_values[][3], int he_atoms,
       double c_r_values[][3], int c_atoms
   ) {
71
```

```
    // Calculate center of mass
    double center_of_mass[3];
    {
        double temp[3] = {0.0, 0.0, 0.0};
        for (int i = 0; i < c_atoms; i++) {
            temp[0] += c_r_values[i][0];
            temp[1] += c_r_values[i][1];
            temp[2] += c_r_values[i][2];
        }
        center_of_mass[0] = temp[0] / c_atoms;
        center_of_mass[1] = temp[1] / c_atoms;
        center_of_mass[2] = temp[2] / c_atoms;
    }

    // Move carbon atoms
    for (int i = 0; i < c_atoms; i++) {
        c_r_values[i][0] -= center_of_mass[0];
        c_r_values[i][1] -= center_of_mass[1];
        c_r_values[i][2] -= center_of_mass[2];
    }

    // Move helium atoms
    for (int i = 0; i < he_atoms; i++) {
        he_r_values[i][0] -= center_of_mass[0];
        he_r_values[i][1] -= center_of_mass[1];
        he_r_values[i][2] -= center_of_mass[2];
    }

    return;

};



int main(int argc, char **argv) {

////////////////////////////////////////////////////////////////////////////////
// Read command arguments
//

    if (argc != 2) {
        std::cout << "Input not understood!\n";
        return 1;
    }
    strcpy(filename_prefix, argv[1]);
    std::cout << "input filename: " << filename_prefix << "\n";

////////////////////////////////////////////////////////////////////////////////
// Read file
//
    // Open raw input file
    strcpy(filename, filename_prefix);
    std::ifstream input_file(filename, std::ios::in | std::ios::binary);
    if (input_file.fail()) {
        std::cout << "Failed to open input file." << std::endl;
        return 1;
    }

    // Read header info; version name
    input_file.read(input_buffer, VERSION_NAME_LENGTH);
    input_buffer[VERSION_NAME_LENGTH] = '\0';
    std::cout << "Version name:" << input_buffer << "\n";

    // Read header info; simulation parameters
    double temperature;
    int32_t he_trotter_n;
    int32_t c_trotter_n;
    int32_t he_atoms;
    int32_t datapoints;
    int32_t simtype;
    int32_t simtime;

    input_file.read((char*) &temperature, sizeof(temperature));
    input_file.read((char*) &he_trotter_n, sizeof(he_trotter_n));
    input_file.read((char*) &c_trotter_n, sizeof(c_trotter_n));
    input_file.read((char*) &he_atoms, sizeof(he_atoms));
    input_file.read((char*) &datapoints, sizeof(datapoints));
    input_file.read((char*) &simtype, sizeof(simtype));
    input_file.read((char*) &simtime, sizeof(simtime));

    std::cout << "temperature: " << temperature << "\n";
    std::cout << "he_trotter_n: " << he_trotter_n << "\n";
```

```
    std::cout << "c_trotter_n:␣" << c_trotter_n << "\n";
    std::cout << "he_atoms:␣" << he_atoms << "\n";
    std::cout << "datapoints:␣" << datapoints << "\n";
    std::cout << "simtype:␣" << simtype << "\n";
    std::cout << "simtime:␣" << simtime << "\n";

////////////////////////////////////////////////////////////////////////////////
//
//

    // Test if supported
    if (!contains(supported_simulations, simtype)) {
        std::cout << "This␣simulation␣type␣is␣not␣supported." << std::endl;
        return 0;
    }


    bool simtype_is_full;
    if (contains(full_simulations, simtype)) {
        simtype_is_full = true;
    } else {
        simtype_is_full = false;
    }

////////////////////////////////////////////////////////////////////////////////
// Open output files:
//

    // Helium position distribution
    strcpy(filename, filename_prefix);
    strcat(filename, "_he_dist.ts");
    std::ofstream outfile_he_dist(filename, std::ios::out | std::ios::binary);

    // Carbon position distribution
    std::ofstream outfile_c_dist;
    if (simtype_is_full) {
        strcpy(filename, filename_prefix);
        strcat(filename, "_c_dist.ts");
        outfile_c_dist.open(filename, std::ios::out | std::ios::binary);
    }

    // Helium-Helium shift
    std::ofstream outfile_he_he_shift;
    if (he_atoms == 2) {
        strcpy(filename, filename_prefix);
        strcat(filename, "_he_he_shift.ts");
        outfile_he_he_shift.open(filename, std::ios::out | std::ios::binary);
    }

    // Helium-Carbon shift
    strcpy(filename, filename_prefix);
    strcat(filename, "_he_c_shift.ts");
    std::ofstream outfile_he_c_shift(filename, std::ios::out | std::ios::binary);

    // Total shift
    strcpy(filename, filename_prefix);
    strcat(filename, "_total_shift.ts");
    std::ofstream outfile_total_shift(filename, std::ios::out | std::ios::binary);



////////////////////////////////////////////////////////////////////////////////
// Calculate time series
//

    // Iterate
    for (int i = 0; i < datapoints; i++) {
        // Read one record from file
        input_file.read((char*) he_r_values, 3 * he_atoms * sizeof(double));
        if (simtype_is_full) {
            input_file.read((char*) c_r_values, 3 * C_ATOMS_C60 * sizeof(double));
        }

        // Move to center of mass co-ordinates (assume mass_he = 0)
        move_to_center_of_mass(he_r_values, he_atoms, c_r_values, 60);

        // Calculate he_dist
        double he_dist = calculate_dist(he_r_values[0]);
        outfile_he_dist.write((char *) &he_dist, sizeof(he_dist));

        // Calculate c_dist
        if (simtype_is_full) {
            double c_dist = calculate_dist(c_r_values[0]);
```

```
            outfile_c_dist.write((char *) &c_dist, sizeof(c_dist));
        }

        // Calculate shifts
        double he_he_shift;
        if (he_atoms == 2) {
            he_he_shift = calculate_he_he_shift(he_r_values);
            outfile_he_he_shift.write((char *) &he_he_shift, sizeof(he_he_shift));
        } else {
            he_he_shift = 0.0;
        }

        double he_c_shift;
        if (simtype == HE_ONLY_SIM) {
            he_c_shift = calculate_he_c_shift_stype1(he_r_values[0]);
        } else if (simtype == CAVITY_SIM) {
            he_c_shift = calculate_he_c_shift_stype2(he_r_values[0]);
        } else if (simtype == FULL_NEW_NEUTRAL) {
            he_c_shift = calculate_he_c_shift_stype4(he_r_values[0], c_r_values);
        } else if (simtype == FULL_NEW_ANION) {
            he_c_shift = calculate_he_c_shift_stype5(he_r_values[0], c_r_values);
        }
        double total_shift = he_he_shift + he_c_shift;

        outfile_he_c_shift.write((char *) &he_c_shift, sizeof(he_c_shift));
        outfile_total_shift.write((char *) &total_shift, sizeof(total_shift));
    }




    return 0;
}
```

Contents of file chem_shift.cpp:

```
#include "distance.cpp"
#include "constants.cpp"

double elj_evaluator(const double r, const double* const c_array,
    const int first, const int last) {

    const int number_of_constants = last - first + 1;

    if (number_of_constants < 1) {
        return 0;
    }

    double factor = ANGS_TO_BOHR / r;
    double value = c_array[number_of_constants - 1];
    for (int i = number_of_constants - 2; i >= 0; i--) {
        value = value * factor + c_array[i];
    }

    return value * pow(factor, first);
}


////////////////////////////////////////////////////////////////////////////
// Shift functions
//


double shift_func_he_c_anion(const double r) {
    const int first = 6;
    const int last = 15;
    const double c_array[last - first + 1] = {
        -38604.94618,
        293275.898,
        -804409.5412,
        742986.3945,
        334207.0458,
        -447632.1219,
        -546557.8469,
        -130158.8838,
        354203.5205,
        656315.6281
    };
    return elj_evaluator(r, c_array, first, last);
}
```

```
double shift_func_he_c_neutral(const double r) {
    const int first = 6;
    const int last = 11;
    const double c_array[last - first + 1] = {
        -3222.162745,
        23143.56522,
        -59215.76864,
        47900.59519,
        33779.15383,
        -52767.09898
    };
    return elj_evaluator(r, c_array, first, last);
}


double shift_func_he_fullerene(const double r) {
    double factor, expot;

    factor = r / ANGS_TO_BOHR;
    expot = 0.292282 * factor;
    expot = (expot - 0.527615) * factor;
    expot = (expot + 0.30585) * factor;
    expot = (expot - 0.0497113) * factor;
    expot = (expot - 0.0015786);
    return -3.47804 * std::pow(factor, -expot);
}

double shift_func_he_he(const double r) {
    double factor, value;

    factor = std::pow(ANGS_TO_BOHR / r, 2);
    value = -246.725 * factor;
    value = (value + 223.978) * factor;
    value = (value - 50.1457) * factor;
    return (value + 4.35223) * pow(factor, 3);
}

////////////////////////////////////////////////////////////////////////////
// He-He shift
//

double calculate_he_he_shift(const double he_r_values[][3]) {
    return shift_func_he_he(calculate_dist(he_r_values[0], he_r_values[1]));
}

////////////////////////////////////////////////////////////////////////////
// He-C shift, simtype 1: HE_ONLY_SIM
//

double calculate_he_c_shift_stype1(const double he_r[3]) {
    double temp = 0.0;
    for (int i = 0; i < C_ATOMS_C60; i++) {
        temp += shift_func_he_c_neutral(calculate_dist(he_r, FULLERENE_R_VALUES[i]));
    }
    return temp;
}

////////////////////////////////////////////////////////////////////////////
// He-C shift, simtype 2: CAVITY_SIM
//

double calculate_he_c_shift_stype2(const double he_r[3]) {
    return shift_func_he_fullerene(calculate_dist(he_r));
}

////////////////////////////////////////////////////////////////////////////
// He-C shift, simtype 4: FULL_NEW_NEUTRAL

double calculate_he_c_shift_stype4(const double he_r[3], const double c_r_values[C_ATOMS_C60][3]) {
    double temp = 0.0;
    for (int i = 0; i < C_ATOMS_C60; i++) {
        temp += shift_func_he_c_neutral(calculate_dist(he_r, c_r_values[i]));
    }
    return temp;
}


////////////////////////////////////////////////////////////////////////////
// He-C shift, simtype 5: FULL_NEW_ANION

double calculate_he_c_shift_stype5(const double he_r[3], const double c_r_values[C_ATOMS_C60][3]) {
    double temp = 0.0;
    for (int i = 0; i < C_ATOMS_C60; i++) {
```

```
        temp += shift_func_he_c_anion(calculate_dist(he_r, c_r_values[i]));
    }
    return temp;
}
```

Contents of file distance.cpp:

```
#include "constants.cpp"
#include <cmath>

#ifndef DISTANCE_MODULE
#define DISTANCE_MODULE

double calculate_dist(const double r_values[3]) {
    int i;
    double temp;

    temp = 0.0;
    for (i = 0; i < 3; i++) {
        temp += r_values[i] * r_values[i];
    }

    return std::sqrt(temp);
}

double calculate_dist(const double r_values_1[3], const double r_values_2[3]) {
    int i;
    double temp, diff;

    temp = 0.0;
    for (i = 0; i < 3; i++) {
        diff = r_values_1[i] - r_values_2[i];
        temp += diff * diff;
    }

    return std::sqrt(temp);
}

#endif
```

Contents of file constants.cpp:

```
#ifndef CONSTANTS_MODULE
#define CONSTANTS_MODULE

#define C_ATOMS_C60 60
#define C_ATOMS_C70 70
#define ANGS_TO_BOHR 1.8897261254578281


////////////////////////////////////////////////////////////////////////////
// Carbon atom positions in fullerene
//

const double FULLERENE_R_VALUES[C_ATOMS_C60][3] = {
     {0.724251, -2.229014,  6.310147},
     {2.343724,  0.000000,  6.310147},
     {0.724251,  2.229014,  6.310147},
    {-1.896113,  1.377606,  6.310147},
    {-1.896113, -1.377606,  6.310147},
     {3.316300,  5.748493,  1.126049},
     {0.695936,  6.599900,  1.126049},
    {-0.475926,  5.748493,  3.469773},
     {1.420187,  4.370886,  4.918274},
     {3.763911,  4.370886,  3.469773},
    {-4.442348,  4.930370,  1.126049},
    {-6.061822,  2.701356,  1.126049},
    {-5.614210,  1.323750,  3.469773},
    {-3.718098,  2.701356,  4.918274},
    {-2.993847,  4.930370,  3.469773},
    {-6.061822, -2.701356,  1.126049},
    {-4.442348, -4.930370,  1.126049},
    {-2.993847, -4.930370,  3.469773},
    {-3.718098, -2.701356,  4.918274},
    {-5.614210, -1.323750,  3.469773},
     {0.695936, -6.599900,  1.126049},
     {3.316300, -5.748493,  1.126049},
     {3.763911, -4.370886,  3.469773},
     {1.420187, -4.370886,  4.918274},
    {-0.475926, -5.748493,  3.469773},
     {6.491934, -1.377606,  1.126049},
```

```
     {6.491934,  1.377606,  1.126049},
     {5.320072,  2.229014,  3.469773},
     {4.595822,  0.000000,  4.918274},
     {5.320072, -2.229014,  3.469773},
     {5.614210,  1.323750, -3.469773},
     {3.718098,  2.701356, -4.918274},
     {2.993847,  4.930370, -3.469773},
     {4.442348,  4.930370, -1.126049},
     {6.061822,  2.701356, -1.126049},
     {0.475926,  5.748493, -3.469773},
    {-1.420187,  4.370886, -4.918274},
    {-3.763911,  4.370886, -3.469773},
    {-3.316300,  5.748493, -1.126049},
    {-0.695936,  6.599900, -1.126049},
    {-5.320072,  2.229014, -3.469773},
    {-4.595822,  0.000000, -4.918274},
    {-5.320072, -2.229014, -3.469773},
    {-6.491934, -1.377606, -1.126049},
    {-6.491934,  1.377606, -1.126049},
    {-3.763911, -4.370886, -3.469773},
    {-1.420187, -4.370886, -4.918274},
     {0.475926, -5.748493, -3.469773},
    {-0.695936, -6.599900, -1.126049},
    {-3.316300, -5.748493, -1.126049},
     {2.993847, -4.930370, -3.469773},
     {3.718098, -2.701356, -4.918274},
     {5.614210, -1.323750, -3.469773},
     {6.061822, -2.701356, -1.126049},
     {4.442348, -4.930370, -1.126049},
     {1.896113, -1.377606, -6.310147},
    {-0.724251, -2.229014, -6.310147},
    {-2.343724,  0.000000, -6.310147},
    {-0.724251,  2.229014, -6.310147},
    {1.896113,  1.377606, -6.310147}
};

#endif
```

B.3 Time series analysis program

Contents of file ts_ana.cpp:

```
#include <iostream>
#include <cstring>
#include <ios>
#include <fstream>
#include <cmath>

#include "stat_ineff.cpp"
#include "autocorrelation.cpp"


#include "json/json.h"
#include "json/json-forwards.h"
#include "jsoncpp.cpp"

#define STRING_LENGTH 4096

char filename_prefix[STRING_LENGTH];
char filename[STRING_LENGTH];

class Histogram {
    private:
        double x1, x2;
        int bins;
        double dx;
        int *hits;

    public:
        Histogram(double x1_, double x2_, int bins_) {
            this->x1 = x1_;
            this->x2 = x2_;
            this->bins = bins_;
            this->dx = (x2_ - x1_) / bins_;
            hits = (int*) malloc(sizeof(*hits) * bins_);
            for (int i = 0; i < bins_; i++) {
                hits[i] = 0;
            }
```

```
        }

        void add_data(double x) {
            int index = floor((x - this->x1) / this->dx);

            if (index >= 0 && index < bins) {
                hits[index]++;
            }
        }


        Json::Value json_serialize() const {
            Json::Value json_data(Json::objectValue);

            json_data["x1"] = this->x1;
            json_data["x2"] = this->x2;
            json_data["bins"] = this->bins;
            json_data["hits_array"] = Json::Value(Json::arrayValue);
            for (int i = 0; i < this->bins; i++) {
                json_data["hits_array"][i] = hits[i];
            }

            return json_data;

        }
};

class TimeSeries {
    public:
        static const int FAIL = 1;
        static const int SUCCESS = 0;

    private:
        double mean;
        double variance;
        double *data;
        double *shifted_data;
        double *autocorrelation_data;
        int datapoints;

    public:
        int load(const char *filename) {
            std::ifstream infile;

            // Open file
            infile.open(filename, std::ios::in | std::ios::binary);
            if (infile.fail()) {
                return this->FAIL;
            }

            // Get file size
            infile.seekg(0, std::ios_base::end);
            this->datapoints = infile.tellg() / sizeof(double);
            infile.seekg(0, std::ios_base::beg);
            std::cout << "datapoints: " << this->datapoints << "\n";

            // Reserve memory
            this->data = (double*) malloc(datapoints * sizeof(double));

            // Load data
            infile.read((char*) this->data, sizeof(double) * datapoints);

            // Initialize
            this->calculate_mean();
            this->make_shifted_data();
            this->calculate_variance();
            this->make_autocorrelation_data();

            return this->SUCCESS;
        }
        double get_moment(int i) const {
            if (i <= 0) {
                return -1;
            }
            if (i == 1) {
                return this->mean;
            }
            if (i == 2) {
                return this->variance;
            }
            return calculate_moment(i);
        }
        double get_inefficiency() const {
```

```
            return calculate_inefficiency(this->data, this->datapoints);
        }

        Histogram get_histogram() {
            const double MULTIPLIER = 10;
            const double BINS = 10000;

            double spread = std::sqrt(this->variance) * MULTIPLIER;
            double x1 = this->mean - spread;
            double x2 = this->mean + spread;

            Histogram hist(x1, x2, BINS);
            for (int i = 0; i < this->datapoints; i++) {
                hist.add_data(this->data[i]);
            }

            return hist;
        }
        std::vector<double> get_autocorrelation_data() const {
            std::vector<double> ac_data;
            for (int i = 0; i < this->datapoints; i++) {
                ac_data.push_back(this->autocorrelation_data[i]);
            }
            return ac_data;
        }

        double get_integrated_autocorrelation_time3() const {
            const int integration_length = 1000;

            // Integrate
            double tau_int = 0.5;
            for (int i = 1; i < integration_length + 1; i++) {
                tau_int += this->autocorrelation_data[i];
            }

            return  tau_int;
        }

    private:
        void calculate_mean() {
            double temp_dp = 0.0;
            for (int i = 0; i < this->datapoints; i++) {
                temp_dp += this->data[i];
            }
            this->mean = temp_dp / this->datapoints;
            return;
        }

        void make_shifted_data() {
            this->shifted_data = (double*) malloc(datapoints * sizeof(double));
            for (int i = 0; i < this->datapoints; i++) {
                this->shifted_data[i] = this->data[i] - this->mean;
            }
            return;
        }

        void make_autocorrelation_data() {
            this->autocorrelation_data = (double*) malloc(sizeof(double) *
                this->datapoints);

            fast_autocorrelation2(this->shifted_data, this->autocorrelation_data,
                this->datapoints);

            double scale = 1 / (this->variance * this->datapoints);
            for (int i = 0; i < this->datapoints; i++) {
                this->autocorrelation_data[i] *= scale;
            }

            return;
        }

        void calculate_variance() {
            double temp_dp = 0.0;
            for (int i = 0; i < this->datapoints; i++) {
                temp_dp += this->shifted_data[i] * this->shifted_data[i];
            }
            this->variance = temp_dp / this->datapoints;
            return;
        }

        double calculate_moment(int n) const {
            double temp_dp = 0.0;
            for (int i = 0; i < this->datapoints; i++) {
```

```
                temp_dp += pow(this->shifted_data[i], n);
            }

            return temp_dp / this->datapoints / pow(this->variance, n / 2.0);
        }

};

int main(int argc, char *argv[]) {
    std::ofstream results_file;
    std::ofstream histogram_file;
    std::ofstream autocorrelation_file;
    Json::Value results_json(Json::objectValue);
    TimeSeries ts;

    // Read command arguments
    if (argc != 2) {
        std::cout << "Input not understood!\n";
        return 1;
    }

    strcpy(filename_prefix, argv[1]);
    std::cout << "filename_prefix: " << filename_prefix << "\n";

    // Load data
    strcpy(filename, filename_prefix);
    strcat(filename, ".ts");
    if (ts.load(filename) != ts.SUCCESS) {
        std::cout << "Failed to read file!\n";
        return 0;
    };

    // Open output files; results:
    strcpy(filename, filename_prefix);
    strcat(filename, ".ana");
    results_file.open(filename);

    // Open output files; histogram:
    strcpy(filename, filename_prefix);
    strcat(filename, ".ana_hist");
    histogram_file.open(filename);

    // Open output files; autocorrelation:
    strcpy(filename, filename_prefix);
    strcat(filename, ".ana_ac");
    autocorrelation_file.open(filename, std::ios::out | std::ios::binary);

    // Write results
    results_json["mean"] = ts.get_moment(1);
    results_json["variance"] = ts.get_moment(2);
    results_json["skew"] = ts.get_moment(3);
    results_json["kurtosis"] = ts.get_moment(4);
    results_json["statistical_inefficiency"] = ts.get_inefficiency();
    results_json["integrated_ac_time"] = ts.get_integrated_autocorrelation_time3();
    results_file << results_json;

    // Write histogram
    histogram_file << ts.get_histogram().json_serialize();

    // Write autocorrelation data
    std::vector<double> ac_data = ts.get_autocorrelation_data();
    for (std::vector<double>::iterator iterator = ac_data.begin(); iterator != ac_data.end(); iterator++) {
        autocorrelation_file.write((const char*) &*iterator, sizeof(double));
    }

    // Close files
    results_file.close();
    histogram_file.close();
    autocorrelation_file.close();
    return 0;
}
```

Contents of file stat_ineff.cpp:

```
#include <cmath>
#include <vector>

// Checks if the given integer is a prime or not
bool isprime(int n) {
    int k;
    if (n == 2) {
        return true;
```

```
    }
    if (n % 2 == 0) {
        return false;
    }

    k = 3;
    while (k*k <= n) {
        if (n % k == 0) {
            return false;
        }
        k += 2;
    }
    return true;
}

// Allows iteration over all primes in a given interval defined by max_int_ and
// min_int_
class PrimeGenerator {
    private:
        int min_int;
        int max_int;
        int current_int;
        int next_int;

    public:
        PrimeGenerator() {
            this->min_int = 0;
            this->max_int = 0;
            this->current_int = 0;
            this->next_int = 0;
        }

        PrimeGenerator(int min_int_, int max_int_) {
            init(min_int_, max_int_);
        }

        void step() {
            this->current_int = this->next_int;
            this->next_int = this->get_next();
            return;
        }

        void init(int min_int_, int max_int_) {
            if (min_int_ < 2) {
                this->min_int = 2;
            } else {
                this->min_int = min_int_;
            }
            this->max_int = max_int_;
            this->current_int = min_int;
            this->next_int = this->get_next();
            return;
        }


        int iteration_ended() {
            return (current_int > max_int);
        }

        int value() {
            return current_int;
        }

    private:
        int get_next() {
            int value;
            if (this->current_int == 2) {
                return 3;
            }

            value = current_int;
            while (1) {
                value += 2;
                if (isprime(value)) {
                    return value;
                }
            }
        }
};

// Performs "block averaging" of data
void reduce_data(
    const double * const data,
```

```
    double *reduced_data,
    const int data_length,
    const int reduction_factor
) {
    for (int i = 0; i < data_length / reduction_factor; i++) {
        double temp_dp = 0.0;
        for (int j = 0; j < reduction_factor; j++) {
            temp_dp += data[i * reduction_factor + j];
        }
        reduced_data[i] = temp_dp / reduction_factor;
    }
}


class ResultPoint {
    public:
        bool is_set;
        double value;

    public:
        ResultPoint() {
            is_set = false;
        }
        void set(double val) {
            this->value = val;
            is_set = true;
        }
        void unset() {
            this->is_set = false;
        }

};

// Helper function for fast_block_average_map
double calculate_variance(const double * const data, const int datapoints) {
    double temp_dp = 0.0;
    for (int i = 0; i < datapoints; i++) {
        temp_dp += data[i];
    }
    double mean = temp_dp / datapoints;

    temp_dp = 0.0;
    for (int i = 0; i < datapoints; i++) {
        temp_dp += pow(data[i] - mean, 2);
    }
    return temp_dp / datapoints;
}

// Helper function for fast_block_average_map
void evaltree(
    const int max_value,
    const int max_multiplier,
    ResultPoint * const results,
    const double * const data,
    const int datapoints,
    const int multiplier,
    const int result_index
) {
    // Evaluate this data
    results[result_index].set(calculate_variance(data, datapoints));

    // Reserve memory for child
    double *reduced_data = (double *) malloc(sizeof(double) * datapoints / 2);

    // Evaluate child data
    for (
        PrimeGenerator pg(
            multiplier,
            std::min(max_value / result_index, max_multiplier)
        );
        !pg.iteration_ended();
        pg.step()
    ) {
        reduce_data(data, reduced_data, datapoints, pg.value());
        evaltree(
            max_value,
            max_multiplier,
            results,
            reduced_data,
            datapoints / pg.value(),
            pg.value(),
            pg.value() * result_index
        );
```

```
    }

    // Free memory
    free((void *) reduced_data);
    return;
}

// Helper function for calculate_inefficiency
void fast_block_average_map(
    const double * const data,
    const int max_block_size,
    const int datapoints,
    const int max_multiplier,
    std::vector<int> *block_sizes,
    std::vector<double> *variances)
{
    ResultPoint *results;

    results = (ResultPoint*) malloc(sizeof(ResultPoint) * (max_block_size + 1));


    for (int i = 0; i < max_block_size + 1; i++) {
        results[i].unset();
    }


    evaltree(max_block_size, max_multiplier, results, data, datapoints, 1, 1);
    for (int i = 0; i < max_block_size + 1; i++) {
        if (results[i].is_set) {
            block_sizes->push_back(i);
            variances->push_back(results[i].value);
        }
    }

    free((void*) results);
    return;
}

// This function does a simple least squares fit. The x, y points are given in
// the vectors "xv" and "yv". The function will fit a line (y = a * x + b) to
// this data. The optimized parameters are saved to the "a" and "b" arguments.
void line_fit(
    const std::vector<double> xv,
    const std::vector<double> yv,
    double &a,
    double &b
) {
    // The number of datapoints read from xv
    int datapoints = xv.size();

    // First intermediate step
    double temp_dp1 = 0.0;
    double temp_dp2 = 0.0;
    double temp_dp3 = 0.0;
    double temp_dp4 = 0.0;
    for (int i = 0; i < datapoints; i++) {
        temp_dp1 += xv[i];
        temp_dp2 += yv[i];
        temp_dp3 += xv[i] * yv[i];
        temp_dp4 += xv[i] * xv[i];
    }

    // Second intermediate step
    double ex = temp_dp1 / datapoints;
    double ey = temp_dp2 / datapoints;
    double exy = temp_dp3 / datapoints;
    double exx = temp_dp4 / datapoints;

    // Save results
    a = (exy - ex * ey) / (exx - ex * ex);
    b = ey - a * ex;
    return;
}

// Smoothens data by binning it.
// Note that length of xv_new and yv_new may not correspond to number bins since
// bins with zero hits are left out
void smoothen_data(
    std::vector<double> xv, std::vector<double> yv,
    std::vector<double> *xv_new, std::vector<double> *yv_new,
    double x1, double x2, int bins
) {

```

```
    double dx;
    double yv_accumulator[bins];
    int yv_hits[bins];
    int datapoints;
    int i, index;

    datapoints = xv.size();

    dx = (x2 - x1) / bins;
    for (i = 0; i < bins; i++) {
        yv_accumulator[i] = 0.0;
        yv_hits[i] = 0;
    }

    for (i = 0; i < datapoints; i++) {
        index = (int) ((xv[i] - x1) / dx + 1) - 1;
        if (0 <= index && index < bins) {
            yv_accumulator[index] += yv[i];
            yv_hits[index] += 1;
        }
    }

    for (i = 0; i < bins; i++) {
        if (yv_hits[i] != 0) {
            xv_new->push_back(x1 + (i + 0.5) * dx);
            yv_new->push_back(yv_accumulator[i] / yv_hits[i]);
        }
    }

}

// This  function estimates the statistical inefficiency in given data. It takes
// as input the data as an array of doubles. The size of that array is given in
// the "datapoints" argument. It returns the estimated statistical inefficiency.
double calculate_inefficiency(const double * const data, const int datapoints) {

    // Calculate the variance of the data
    double variance = calculate_variance(data, datapoints);

    // Calculate the variances of block averaged datas
    std::vector<int> block_sizes; // Save the block sizes here
    std::vector<double> variances; // Save the corresponding variances here
    fast_block_average_map(data, datapoints / 10, datapoints, 5, &block_sizes,
        &variances);
    int ineff_datapoints = block_sizes.size();

    // Calculate the inverse square roots of the block sizes and the
    // corresponding statistical inefficiency estimates
    std::vector<double> ineff_values;
    std::vector<double> inverse_sqrt_block_sizes;
    for (int i = 0; i < ineff_datapoints; i++) {
        ineff_values.push_back(variances[i] * block_sizes[i] / variance);
        inverse_sqrt_block_sizes.push_back(pow(block_sizes[i], -0.5));
    }

    // Estimate the statistical inefficiency by fitting a line to the estimates
    double a, b;
    std::vector<double> xv, yv;
    smoothen_data(inverse_sqrt_block_sizes, ineff_values, &xv, &yv, 0.01, 1.0,
        100);
    line_fit(xv, yv, a, b);

    // Return the constant part of line fit, that approximates statistical
    // inefficiency.
    return b;
}
```

Contents of file autocorrelation.cpp:

```
#include <fftw3.h>

void fast_autocorrelation2(double *data_in, double *data_out, int data_length) {
    fftw_complex *fft_data;
    fftw_plan fft_forward_plan;
    fftw_plan fft_backward_plan;
    const int padded_data_size = 2 * data_length - 1;

    // Allocate memory
    fft_data = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * padded_data_size);

    // Make plans
    fft_forward_plan = fftw_plan_dft_1d(padded_data_size, fft_data, fft_data, FFTW_FORWARD, FFTW_ESTIMATE);
```

```
    fft_backward_plan = fftw_plan_dft_1d(padded_data_size, fft_data, fft_data, FFTW_BACKWARD, FFTW_ESTIMATE);

    // Initialize fft data
    for (int i = 0; i < data_length; i++) {
        fft_data[i][0] = data_in[i];
        fft_data[i][1] = 0;
    }
    for (int i = data_length; i < padded_data_size; i++) {
        fft_data[i][0] = 0;
        fft_data[i][1] = 0;
    }

    // Compute forward fft
    fftw_execute(fft_forward_plan);

    // Multiply with complex conjugate
    for (int i = 0; i < padded_data_size; i++) {
        double a = fft_data[i][0];
        double b = fft_data[i][1];
        fft_data[i][0] = (a * a + b * b);
        fft_data[i][1] = 0;
    }

    // Compute backward fft
    fftw_execute(fft_backward_plan);

    // output data
    for (int i = 0; i < data_length; i++) {
        data_out[i] = fft_data[i][0] / padded_data_size;
    }

    // delete plans
    fftw_destroy_plan(fft_forward_plan);
    fftw_destroy_plan(fft_backward_plan);

    // Free memory
    fftw_free(fft_data);

    return;
}
```